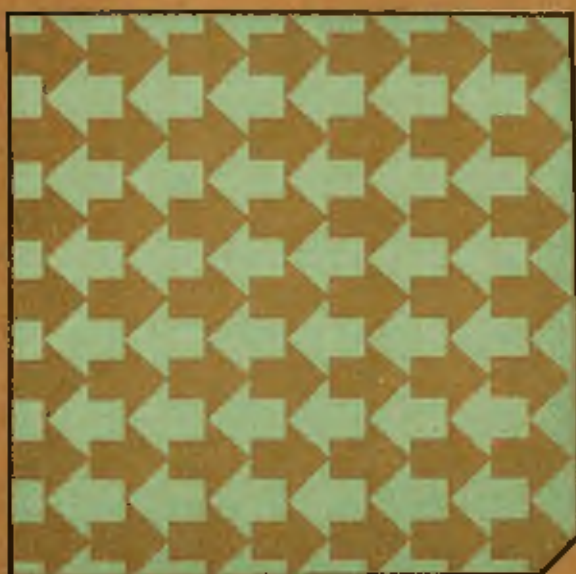
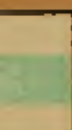


**МАТЕМАТИЧЕСКОЕ
ОБЕСПЕЧЕНИЕ
ЭВМ**

М. Сингер

**МИНИ-ЭВМ RDP-11:
ПРОГРАММИРОВАНИЕ
НА ЯЗЫКЕ АССЕМБЛЕРА
И ОРГАНИЗАЦИЯ
МАШИНЫ**





PDP-11

ASSEMBLER LANGUAGE
PROGRAMMING
AND MACHINE
ORGANIZATION

Michael Singer

Stanford University

JOHN WILEY & SONS
NEW YORK • CHICHESTER • BRISBANE •
TORONTO • SINGAPORE
1980

**МАТЕМАТИЧЕСКОЕ
ОБЕСПЕЧЕНИЕ
ЭВМ**

М. Сингер

**МИНИ-ЭВМ PDP-11:
ПРОГРАММИРОВАНИЕ
НА ЯЗЫКЕ АССЕМБЛЕРА
И ОРГАНИЗАЦИЯ
МАШИНЫ**

Перевод с английского
А. Ю. Каргашина и
А. С. Миркотан

под редакцией
Ю. М. Баяковского

ББК 22.19

С 38

УДК 681.3

Сингер М.

С 38 **Мини-ЭВМ PDP-11; Программирование на языке ассем-
блера и организация машины: Пер. с англ.— М.: Мир,
1984.— 272 с., ил.**

Книга американского специалиста представляет собой подробное учебное пособие по программированию на языке ассемблера для машины PDP-11, послужившей прототипом для отечественных мини-ЭВМ СМ-3, СМ-4 и др. Рассмотрены такие вопросы, как ввод-вывод, механизм прерываний, управление памятью. Книга содержит многочисленные примеры и упражнения.
Для всех, кто работает с мини-ЭВМ.

С $\frac{2405000000-424}{041(01)-84}$ 164-84, ч. 1

ББК 22.19
518

Редакция литературы по математическим наукам

© 1980 by John Wiley & Sons, Inc.
All Rights Reserved. Authorized translation
from English language edition published by
John Wiley & Sons, Inc.

© Перевод на русский язык, «Мир», 1984

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Может показаться, что эта книга переносит нас лет на 25 назад, во вторую половину 50-х годов, когда только начали появляться языки высокого уровня, а о языках очень высокого уровня не было еще и речи, когда программист прекрасно ориентировался в архитектуре *своей* машины, наизусть знал все ее коды команд и когда восьмеричной системе счисления прочили блестящее будущее. Иллюзия подкрепляется тем, что совсем недавно в центре внимания была идея концентрации в одной установке больших вычислительных мощностей и их *коммунальное* потребление в режиме разделения времени. И вот все неожиданно и быстро изменилось: на первом плане оказались *изолированные* сначала мини-, затем микро- и, наконец, персональные ЭВМ.

Эти современные малогабаритные машины уступают своим сестрам-монстрам из 50-х годов разве что в размерах, превосходя их во многих других отношениях. На современных мини-ЭВМ есть операционные системы и трансляторы с языков высокого уровня. Но тем не менее часто требуется знание архитектуры машины и языка ассемблера. Объясняется это, по-видимому, двумя причинами. Во-первых, как правило, ощущается ограниченность ресурсов: невелика непосредственно адресуемая память, часто в системе команд нет операций умножения и деления и т. д. Во-вторых, многие мини-ЭВМ включаются как элемент в систему управления, работают в реальном масштабе времени, имеют развитую сеть связи с внешней средой. В этих случаях применение языков высокого уровня оказывается затруднительным, а порой и совсем невозможным.

В книге М. Сингера подробно и полно изложены сведения, необходимые при программировании на языке ассемблера и полезные при программировании на языках высокого уровня. Более того, речь в книге идет о машине PDP-11, которая очень распространена в мире и которая послужила прототипом для многих других машин, в частности отечественных.

Машины с архитектурой PDP-11 нашли широкое применение в высших учебных заведениях. Надо сказать, что автор книги работает в Станфордском университете, а сама книга рекомендуется как учебное пособие по курсу программирования для

ЭВМ. Поэтому книга строится в соответствии с определенными методическими установками и существенно отличается от обычной фирменной документации. Испытав на себе все трудности, которые вызывает освоение новой машины по такой документации, я могу высказать убеждение, что многим (и не только начинающим) программистам книга поможет лучше понять свою машину и научиться эффективно ее использовать.

Книгу перевели А. С. Миркотан (предисловие, гл. 1 и 2) и А. Ю. Каргашин (гл. 3, 4 и приложения).

Ю. Баяковский

ПРЕДИСЛОВИЕ

Во всем мире сейчас ни одна коммерческая и научная организация не обходится без малых ЭВМ. Они помогают извлекать выгоду, освобождают от тяжелой нудной работы и, увы, слишком часто служат и источником разочарований. Эта книга поможет пользователям улучшить положение по первым двум пунктам и облегчить его в отношении последнего.

Книга целиком посвящена одной из самых популярных и гибких малых ЭВМ PDP-11, созданной фирмой Digital Equipment Corporation (DEC). Электронная и концептуальная база, вообще говоря, одинакова для всех вычислительных машин, поэтому наше изложение с относительно небольшими изменениями применимо и к другим ЭВМ. Однако мы не собираемся выкинуть на прилавки книжных магазинов еще одно общее руководство по устройству вычислительных машин. Программирование — это набор практических навыков, опирающихся на теоретические знания, но они превращаются в подлинное мастерство лишь в результате постоянного применения на практике. Поскольку новичков, сидящих за терминалами, гораздо больше интересуют конкретные особенности вычислительных машин, а не их структурное сходство, мы считаем, что первое знакомство с машинами должно начинаться с определенной ЭВМ.

Книга построена таким образом, чтобы как можно быстрее научить читателя писать законченные, хотя и достаточно тривиальные, программы. Поэтому с самого начала необходим доступ к PDP-11. Больше никаких условий не ставится; более того, мы не требуем предварительных теоретических знаний или опыта работы на ЭВМ. И начинающие программисты, и квалифицированные пользователи ЭВМ PDP-11 могут читать эту книгу, не прибегая к другим источникам. Она может также служить дополнительным пособием для обычного первого или второго курса программирования.

Работу вычислительной машины, как и деятельность общества, невозможно познать, не постигнув правил и тонкостей ее языка. Язык ЭВМ — это ее машинный код; вычислительная машина не «понимает» ничего другого и будет воспринимать инструкции таких языков высокого уровня, как Бейсик, Кобол,

Фортран и Паскаль, лишь в том случае, если они переведены на машинный язык. Программа перевода, называемая *компилятором*, сама может быть написана на языке высокого уровня. Но на каком-то этапе ЭВМ необходимо растолковать «смысл» операторов языка высокого уровня, воспользовавшись машинным кодом. Таким образом, если мы хотим познать радость наиболее полного общения с PDP-11, нужно понимать ее машинный код. Даже язык ассемблера PDP-11, который, в сущности, не что иное, как набор команд машинного кода, представленных в доступном для чтения формате, образует барьер между пользователем и машиной. Однако этот барьер не так велик, и программирующий на языке ассемблера имеет в своем распоряжении все средства ЭВМ PDP-11; правда, чтобы в полной мере ими воспользоваться, нужно иметь четкое представление о всех деталях машинного кода.

Даже тот, кто обычно не программирует на языке ассемблера, выиграет от более близкого знакомства с работой ЭВМ: никогда нельзя быть уверенным, что язык ассемблера вам ни разу не потребуется. Универсальные языки высокого уровня не отражают особенностей конкретной машины, а как раз эти особенности и могут понадобиться в определенной задаче.

Кроме того, в ряде случаев язык ассемблера может оказаться более эффективным. На малой ЭВМ компилятор с языка высокого уровня не может быть большим, и потому в нем не предусматриваются средства оптимизации, без которых невозможно получить наиболее эффективный код. Таким образом, растет время выполнения программ, а следовательно, и стоимость. Некоторые считают, что это соображение устарело. Выдвигается та точка зрения, что неэффективный метод или *алгоритм* расточительнее, чем неэффективное кодирование, и что бо́льшая ценность времени программиста по сравнению с машинным временем оправдывает даже использование неэффективных алгоритмов. Можно согласиться с тем, что время программиста важнее времени машины, однако у человека оно может уйти и на то, чтобы ждать, пока выполнится плохая программа, написанная другим программистом. Иными словами, если вместо рационального использования трех ЭВМ неэффективно работают четыре, то жизненные ресурсы человека расходуются впустую. Нельзя не признать, что неуклюже закодированный хороший алгоритм, как правило, приводит к лучшим результатам, чем примитивный алгоритм, на который потратил силы самый опытный программист. Однако это всего лишь интуитивные доводы. Чтобы взвесить все эти точки зрения, нужно провести подробный анализ показателя стоимость — эффективность. Ясно, что достоверный анализ может быть проведен только программистом, в совершенстве изучившим все тонкости своего ремесла; попытки

повысить эффективность программы никогда не оправдывают затраченных усилий, если человек не представляет, как это можно сделать.

Хотя книга и не посвящена разработке алгоритмов (фактически термин «алгоритм» не используется за пределами предисловия), тем не менее мы всегда оперируем положениями истинно структурного подхода к программированию. Так, мы не поддались искушению построить книгу на основе демонстрации своих собственных любимых и весьма длинных программ. Вместо этого текст изобилует примерами программ целого ряда небольших характерных задач. Глава 3, ключевая для этой книги, посвящена тем средствам PDP-11, которые позволяют строить законченные программы из отдельных блоков, или *модулей*. По возможности мы старались избегать абстрактных рассуждений, однако тщательно следили за тем, чтобы все изложение пронизывали наиболее важные понятия теории программирования.

Глава 1 знакомит читателя с вычислительной системой. Операционная система затрагивается в той степени, в какой это необходимо для достижения основной цели — заставить работать простую программу. Наше стремление к тому, чтобы из практики постепенно рождалась теория, требует от обучающихся возможности экспериментировать и, в частности, приводит к такой непоследовательности, как описание вывода под управлением монитора на ранних этапах изложения. Полное объяснение механизма обращений к монитору, естественно, отнесено дальше. Пользователи такой PDP-11, которая не имеет операционной системы, могут, чтобы начать работать, читать § 4.1 параллельно с гл. 1; предвидя такую возможность, мы построили этот параграф так, чтобы он мало опирался на более ранние главы.

В гл. 2 широкий диапазон команд языка ассемблера демонстрируется на примерах решения задач. Мы вводим только те команды, которые могут быть тотчас же использованы, поэтому многие команды появляются в книге позже. Тем не менее вся книга охватывает полный набор команд языка ассемблера PDP-11.

На протяжении всей книги наше отношение к операционной системе остается противоречивым. Строго говоря, эта система представляет собой просто набор программ и не имеет ничего общего с машиной как таковой. С другой стороны, среди программ имеются чрезвычайно полезные (как, например, ассемблеры и редакторы), и все они настолько тесно координируют свои действия с работой машины, что поистине могут рассматриваться как ее часть. Поэтому многие аспекты применения PDP-11 описываются как при условии, что операционная система существует, так и при условии, что ее нет. Особенно это касается гл. 4, где мы обсуждаем ввод и вывод для внешних запо-

минающих устройств в обеих ситуациях. В эту главу также входят ввод-вывод с терминала, прерывания и управление памятью.

В книге два приложения. Приложение А содержит описание ODT—средства отладки, имеющегося в операционной системе PDP-11. Его можно читать параллельно с основным текстом; начинать следует при изучении гл. 2.

В приложении Б вводится целая арифметика с повышенной точностью и арифметика с плавающей точкой для PDP-11. Повышенная точность представляет собой обычное требование для мини-ЭВМ с малой длиной слова; приведенные фрагменты подпрограмм докажут это.

Книга насыщена упражнениями, хотя бы часть которых нужно выполнять перед тем, как двигаться дальше. Большинство упражнений являются простыми проверками на понимание, несмотря на то что они весьма различаются по времени, которое затрачивается на решение. Символом «звездочка» * отмечены чуть более сложные задачи.

Многие коллеги оказали поддержку, поделились советами и ценной информацией, доступной узкому кругу; среди них Джон Бенедикт из Ross Systems Inc., Стив Кэллис из DEC и Тэд Панофски из Artificial Intelligence Laboratory, Станфордский университет. Постоянно ощущалась поддержка и одобрение сотрудников издательства John Wiley & Sons.

Майкл Сингер

1. ВВЕДЕНИЕ

1.1. Вычислительная система

Программист, начинающий работать на PDP-11, вряд ли столкнется с этой ЭВМ в чистом виде, без прикрас. Обычно есть *терминал*, через который осуществляется связь с вычислительной машиной. Имеются устройства (как правило, это *диски* или *ленты*), на которых может постоянно храниться информация. Существует несколько разных моделей машины PDP-11; самые большие из них могут обслуживать многих пользователей, находящихся за своими терминалами (в режиме *разделения времени*). В этом случае устройства памяти обычно находятся в машинном зале, вне поля зрения программиста.

Такие устройства (*аппаратные средства*), как терминалы и диски, известны под названием *периферийные*. Периферийные устройства, используемые вместе с PDP-11, весьма разнообразны как по уровню сложности интерфейсов, которые позволяют им взаимодействовать с другими ЭВМ, так и по своему внешнему исполнению. Они считаются периферийными по отношению к шкафу со сложной электронной схемой (быть может, со сверкающей панелью тумблеров и лампочек), которая составляет основу ЭВМ PDP-11 в вашей системе.

Даже если ваша система PDP-11 очень мала и управляется только с одного терминала (*изолированная система*), она, почти наверняка, включает в себя больше, чем мы только что описали. Посмотрим, как работает терминал. Он похож на пишущую машинку, но имеет ряд специфических черт. Существует несколько разных типов терминалов; некоторые выдают вводимые с клавиатуры литеры, а также сообщения машины не на обычный рулон бумаги, а на экран вроде телевизионного. На разных терминалах отдельные специальные литеры расположены в разных местах, поэтому вам придется потратить несколько минут на знакомство с особенностями всякой новой или просто незнакомой модели.

Как и в обычной пишущей машинке, здесь есть клавиша SHIFT¹⁾. Заметим, однако, что многие терминалы имеют только прописные (заглавные) буквы. Это не должно беспокоить про-

¹⁾ В терминалах СМ ЭВМ ей соответствует клавиша ВР (или НР). —
Прим. перев.

граммиста, поскольку в машинных командах прописные и строчные буквы не различаются. В таких терминалах для печатания обычных букв не пользуйтесь клавишей SHIFT, так как иногда при этом могут получиться другие символы. Например, в некоторых терминалах символ @ есть SHIFT-P, символ] — SHIFT-M, а [— SHIFT-K. Обычно эти знаки четко обозначены на соответствующих клавишах.

Будьте внимательны, чтобы не спутать следующие литеры: I (заглавную i), l (строчную L) и 1 (единицу); O (заглавную букву O) и 0 (нуль), а также круглые (...), угловые <..> и квадратные [...] скобки.

Интересной особенностью обладает клавиша CONTROL ¹⁾. Как и SHIFT, она ничего не выполняет сама по себе, однако, нажимая ее одновременно с другими клавишами, мы получаем совершенно иной набор литер. В некоторых случаях это обычные литеры. Так, если нажать CONTROL-A, то на бумаге или экране появится ^A. Если, однако, вы нажмете CONTROL-I, то не появится никакой литеры, но эффект при этом бывает, как правило, такой же, как от нажатия клавиши табуляции. При табуляции литеры обычно печатаются на каждой восьмой позиции. В большинстве систем PDP-11, если нажать CONTROL-C во время выполнения программы, то появится ^C и программа остановится (если вычисления все же продолжаются, то нужны два CONTROL-C). Есть CONTROL-литеры ²⁾, которые вызывают функции «прерывания». (В случае необходимости с ними можно ознакомиться самостоятельно).

В этой книге клавиша CONTROL будет обозначаться с помощью символа ^. *Предупреждение*: это не «крышечка» над буквой и не символ «стрелка вверх», имеющиеся в некоторых клавиатурах (последний символ обычно можно получить, нажав SHIFT-N). Если сначала нажать ^, затем C, то тоже появится ^C, но уже не будет обладать особым свойством CONTROL-C.

В этой книге сочетание типа ^A всегда означает, что, если нет особых оговорок, при нажатой клавише CONTROL вводится некая литера.

Когда переключатель режима *on-line* ³⁾ «включен» и тем самым обеспечена связь между терминалом и машиной, нажмите ^I. Как уже отмечалось, после этого в большинстве систем на терминале будет выполняться табуляция. Теперь отключите режим *on-line*, отсоединив тем самым терминал от ЭВМ. Терминал еще будет функционировать как пишущая машинка, но вы, ве-

¹⁾ В терминалах СМ ЭВМ ей соответствует клавиша УПР.— *Прим. перев.*

²⁾ Часто используется термин «управляющие символы».

³⁾ Иногда этот режим называют оперативным, а *off-line* — автономным,

роятно, обнаружите, что $\wedge I$ теперь ничего не исполняет (все зависит от типа используемого терминала). Некоторые современные терминалы, особенно экранные, обладают встроенным в схему узлом, который осуществляет табуляцию в ответ на $\wedge I$; в других этого нет, и $\wedge I$ не будет работать, если терминал находится в автономном режиме. В оперативном режиме для выполнения табуляции требуется сложный процесс. Пользователь не видит проявлений этого процесса (для пользователя он *прозрачен*). Все, что пользователь вводит с терминала в режиме on-line (пользовательский *ввод*), сразу же попадает в машину. Специальная программа, которая всегда доступна в вычислительной машине, проверяет этот ввод; программа организована так, что в ответ на $\wedge I$ заставляет передвигаться на нужное число позиций печатающую головку или курсор терминала. Когда терминал находится в оперативном режиме, то между клавиатурой и бумагой или экраном не существует прямой связи. Введенный с клавиатуры символ *выводит* на бумагу или экран терминала (т. е. создает эхо литеры) не механическая реакция терминала, а та же самая программа.

Эта программа называется *монитор*. Функции монитора не ограничиваются печатанием литер на терминале. Монитор управляет выполнением пользовательской программы, ищет место на диске для записи программ или данных, организует ввод и вывод и делает многое другое. В режиме разделения времени монитор распределяет время компьютера между разными пользователями. По существу, это управляющая система вычислительной машины.

Как неотъемлемая часть вычислительной системы (но программа, а не физическое устройство) монитор представляет собой пример *программного обеспечения ЭВМ*. Значительная часть работы монитора состоит в управлении другими программами, которые постоянно находятся в машине, выполняя задания пользователей. Это *системные программы*, которые также являются частью программного обеспечения; из них мы, в частности, изучим редакторы, ассемблеры и загрузчики. Монитор и различные системные программы, которыми снабжены устройства вычислительной машины, в совокупности составляют *операционную систему*.

Можно работать на ЭВМ PDP-11, вовсе не пользуясь операционной системой. Однако, как только вы узнаете о ее существовании, то теряете уверенность в собственных силах, и эта работа становится неописуемо нудной. Без операционной системы ввод программы в машину и ее выполнение превращаются в длительный процесс, и при этом каждый раз он состоит примерно из одной и той же последовательности шагов; другими словами, это работа как раз для ЭВМ. Итак, должна быть написана про-

грамма для управления программами; если эта «суперпрограмма» работает постоянно, то мы имеем примитивную операционную систему. В этом случае задача ввода программы в машину выполняется лишь однажды (почему?), после чего управление программой берет на себя суперпрограмма.

Как уже отмечалось, достаточно квалифицированные программисты могут написать свои собственные операционные системы. При этом они бывают организованы так, чтобы полностью отвечать требованиям своих пользователей. Однако уже существует множество операционных систем, предназначенных для PDP-11. Многие пользователи ЭВМ обнаруживают, что их нужды вполне удовлетворяются одной из коммерчески доступных систем, которая готова к работе, как только включают машину. Или же можно приобрести отдельные компоненты и, вооружившись необходимыми знаниями, собрать из них полную систему (*модульная система*). Обычно приобретаемое программное обеспечение записано на диске или магнитной ленте либо отперфорировано на рулоне бумажной ленты; при этом покупатель обязуется использовать его только для своей системы. Системные программы часто оказываются очень сложными, а потому дорогостоящими; отсюда стремление защитить их таким образом. Сейчас пока нельзя сказать, в какой степени программное обеспечение действительно защищено законом о патентах и авторских правах.

Из-за большого разнообразия операционных систем могут возникнуть проблемы, если вам придется работать на PDP-11 где-то в другой организации. В ответ на допустимую последовательность литер, введенных пользователем, программа монитора будет выполнять некоторую совокупность действий; у монитора есть список, с помощью которого он «узнаёт» пользовательские *команды*. Сложность состоит в том, что разные мониторы могут иметь различные списки, а выполнить программу без знания команд монитора данной системы невозможно. Подобные различия затрудняют весь процесс создания программ и могут даже привести к тому, что программа будет прекрасно работать в одной системе и совсем не будет работать в другой.

Разработчики программного обеспечения периодически предлагают новые, усовершенствованные версии; администраторы систем часто пытаются «рационализировать» свои операционные системы, иногда и улучшая их. В первом случае документация редко идет в ногу с изменениями, а если все же успевает за ними, то может быть неточной; во втором случае документация, как правило, вообще отсутствует. Поэтому даже такая операционная система PDP-11, которая считается самой обычной, может таить в себе неожиданности. По этой причине нет смысла подробно описывать здесь какую-либо версию данной системы;

читателю, работающему в другой или усовершенствованной версии, это очень скоро наскучит. С другой стороны, не будет большой пользы и в общих теоретических утверждениях относительно операционных систем, так как это мало поможет студенту, намеревающемуся просто пропустить программу.

Будем строить изложение на одном уровне абстракции, опуская детали какой-либо определенной операционной системы. В этой главе, которая посвящена написанию нашей первой программы для PDP-11 и подготовке ее к счету, используется лишь узкий круг возможностей операционной системы. К счастью, этого объема достаточно для понимания принципов организации большинства систем фирмы DEC. Их мы всегда придерживаемся при обсуждении предполагаемого поведения операционной системы во время выполнения очередной задачи. Наши примеры программ могут и не работать в вашей системе точно в той форме, в которой мы их даем, однако вы уже ознакомитесь с терминологией, с помощью которой сможете со знанием дела пользоваться справочными руководствами по программному обеспечению и в нужный момент быстро осознать происходящее, чтобы принять решение. Например, в § 1.2 мы рассматриваем редактор — системную программу, с помощью которой можно организовать в удобной и доступной форме текст или команды программы, хранящиеся на диске или ленте (в *файле* на диске или ленте). В каждом редакторе есть команда, которая позволяет распечатать следующую строку вашего файла. В редакторах фирмы DEC это, как правило, однобуквенная команда. В редакторе EDIT, на который ориентирован § 1.2, это команда L. Студент, использующий, скажем, редактор TЕСO, должен просто проверить по руководству, что эквивалентной командой в данном случае будет T, и сделать пометку в соответствующем месте книги.

После изучения гл. 1 вы будете достаточно хорошо знать свою операционную систему, чтобы разобраться в остальном тексте вплоть до гл. 4, в которой описывается программирование ввода-вывода. Кроме того, вы сможете без особых затруднений выполнить программу в любой операционной системе фирмы DEC, которая вам попадется.

Почти все ЭВМ PDP-11 снабжены операционными системами фирмы DEC либо системами, разработанными по тем же принципам; лишь немногие имеют систему UNIX производства фирмы Bell Laboratories. Эту систему мы не обсуждаем не только по той причине, что по ней уже есть хорошая документация, но и потому, что UNIX разрабатывалась для использования на многих разных компьютерах и не раскрывает индивидуальных особенностей ЭВМ PDP-11. Мы больше заинтересованы в том, чтобы читатель поближе познакомился с вычислительной машиной

и научился оптимальным образом использовать специфические особенности PDP-11.

Несмотря на первые слова этого параграфа, мы не забыли про студента, сидящего перед небольшой машиной PDP-11 с минимальными возможностями. Действительно, поскольку внедрение LSI-11 (микрокомпьютерной версии PDP-11) открыло доступ к PDP-11 для непрофессионалов, таких читателей будет немало. Если это относится и к вам, то, чтобы подготовиться к счету свою первую программу, после § 1.4 прочтите § 4.1, который преднамеренно был написан так, чтобы как можно меньше зависеть от более ранних параграфов и тем самым обеспечить вам начало работы на машине.

1.2. Редактор

Как указывалось в § 1.1, редактор нужен для того, чтобы помочь при формировании и хранении файлов. Часть ваших файлов будет представлять собой последовательности машинных команд, т. е. *программы*. Другие могут быть наборами данных, предназначенных для обработки с помощью программ.

Поскольку мы пока не знаем, как выглядят машинные команды, мы не сможем написать программу, но тем не менее сумеем сформировать простой файл.

Системы разделения времени. Монитор откроет вам доступ к редактору. В системах разделения времени, однако, монитор сначала проверит, имеете ли вы на это право. Поэтому в начале каждого сеанса связи с ЭВМ или *задания* нужно инициировать процедуру проверки, обратившись к монитору с помощью команды

LOGIN

В некоторых системах ей соответствует команда **HELLO**. Монитор не отреагирует на введенные литеры до тех пор, пока вы не нажмете клавишу возврата каретки **CARRIAGE RETURN**. Это удобно, так как позволяет вовремя выявить все опечатки в тексте. Ошибки можно исправить, если нажать клавишу **RUBOUT** (иногда она называется **DELETE**) и затем перепечатать текст. В некоторых системах вместо этого можно использовать клавишу **BACKSPACE**. Если вы сделаете много ошибок, то, видимо, проще стереть всю строку, которую вы печатаете, и начать сначала. Для этого надо нажать **^U**.

В ответ на **LOGIN** монитор сначала запросит ваш шифр (опознавательный номер), затем пароль. Получив удовлетворитель-

ный ответ, он объявит о готовности принять ваши команды, напечатав соответствующее сообщение. В ряде систем это будет текст: **READY** (готов), в других — просто одна литера, например . (точка) или @. Появление такого сообщения говорит о том, что вы находитесь в *режиме команд монитора*. Помните, что в большинстве систем в этот режим всегда можно вернуться, дважды нажав ^C.

В большинстве систем разделения времени задание будет продолжаться, и расходная статья вашего счета будет расти до тех пор, пока вы каким-то образом не сообщите монитору о желании завершить задание. Для этого надо ввести команду, а не просто выключить терминал и уйти. Обычно это команда **BYE**. В некоторых системах вы получите запрос **CONFIRM** (подтвердите), на который нужно ответить **YES**, затем **CARRIAGE RETURN**.

УПРАЖНЕНИЕ. Попрактикуйтесь в использовании процедуры образования и завершения задания, пока не узнаете ее как следует.

Открытие файла. Чтобы записать файл в режиме команд монитора, нужно монитору дать команду о запуске системной программы редактора. В некоторых системах существует несколько программ-редакторов. Как правило, командой монитора о запуске системной программы является **RUN**; в отдельных системах используется команда **R**. Напечатайте эту команду, затем пробел и имя программы редактора. Например, чтобы вызвать редактор **EDIT**, нужно напечатать

RUN EDIT

затем возврат каретки **CARRIAGE RETURN** (эту клавишу мы будем обозначать символом ↵). В некоторых системах при использовании команды **RUN** имени системной программы должен предшествовать знак доллара \$. В этом случае нужно ввести

RUN \$EDIT↵

В ответ на это редактор сообщит о готовности принять команды, в которых указано, с какими файлами предстоит иметь дело; сообщение может быть в виде одного символа, например #. Поскольку мы хотим образовать новый файл, то должны выбрать для него имя. Длина имени файла может быть до шести литер. Первая литера должна быть буквой, остальные — либо буквами, либо цифрами. У нас есть все основания назвать свой файл **TEST**. Имя файла заканчивается *расширением*, состоящим из четырех литер, первая из которых точка. Расширения имен

файлов служат для сообщения пользователям и системным программистам полезной информации о типе файла. Например, если бы наш файл представлял собой (макро) программу на языке ассемблера, то мы могли бы назвать его **TEST.MAC**. Мы же хотим записать файл, содержащий всего лишь простой текст, а для этого не предусмотрено никакого специального расширения. Однако совсем не указывать расширение тоже плохо, поскольку в ряде систем предполагается, что файл без расширения написан на каком-то выбранном администратором системы языке. Например, если администратор предпочитает язык Бейсик, то нашей программе **TEST** автоматически будет дано расширение *по умолчанию* **TEST.BAS**. Чтобы избежать такой путаницы, подберем подходящее мнемоническое расширение и назовем наш файл **TEST.TXT**.

Мы хотим сообщить редактору, что собираемся послать написанное в файл **TEST.TXT**, т. е. **TEST.TXT** должен быть *открыт для вывода*. Поэтому в ответ на сообщение редактора (в данном случае **#**) о готовности принять такую информацию мы печатаем

TEST.TXT↵

Затем **EDIT**, зная, что вы хотите делать с напечатанным вами текстом, объявит о готовности принять команды редактирования текста, напечатав *****. Не все редакторы отделяют таким образом стадии приема команд открытия файла от стадии приема команд редактирования текста. Например, редактор **TECO** выдает ***** в качестве приглашения для любого типа команд. Поэтому недостаточно напечатать только имя файла; сначала нужно сообщить **TECO** с помощью команды **EW**, что вы собираетесь открыть файл для вывода

EWTEST.TXT

Заметим, что в **TECO** команды вводят при нажатии два раза подряд клавиш **ESCAPE** (в некоторых терминалах она называется **ALTMODE**), а не при помощи **CARRIAGE RETURN**. Когда вы нажмете **ESCAPE**, монитор напечатает на терминале символ **\$**. Редактор **TECO** есть в целом ряде систем **PDP-11**; это очень элегантный и мощный редактор, и мы вам его настоятельно рекомендуем.

Запись в файл. Если редактор готов принять команды редактирования текста, то он отводит рабочую область для того текста, с которым вы хотите работать. Эта рабочая область называется *буфером* и находится в легко доступном месте хранения.

информации, т. е. в *памяти* машины. Память (также известная под названием *запоминающее устройство*) доступна как для программ пользователей, так и для системных программ. Информация, хранящаяся в памяти, не остается там по окончании сеанса связи с ЭВМ; для более длительного хранения ее необходимо записать в файлы на дисках или лентах.

В начале сеанса редактирования буфер пуст. Первое, что надо сделать, это *поместить* туда нужный нам текст. В некоторых редакторах все, что вы на этом этапе напечатаете до команды конца текста, воспринимается как текст для вставки. Однако в более общем случае сначала необходимо ввести специальную команду, например I. Вы должны четко представлять себе, как правильно использовать эту команду (*синтаксис* команды). Например, в TECO синтаксис команды I очень прост: напечатав I, вы затем просто печатаете весь текст, который хотите вставить, после этого нажимаете ESCAPE два раза подряд. Сеанс может выглядеть, скажем, так:

```
*I THE QUICK
  BROWN FOX
  $$
```

Символ * напечатал редактор TECO; все остальное было напечатано пользователем. Вспомните, что \$ появляется на термине после нажатия клавиши ESCAPE. Заметим, что редактор воспринимает CARRIAGE RETURN как часть текста. В результате буфер содержит следующий текст:

```
THE QUICK↵BROWN FOX↵
```

Здесь мы использовали символ ↵, чтобы показать, что в буфере есть возврат каретки.

В редакторе EDIT синтаксис команды I несколько иной. Для того чтобы подготовить этот редактор к помещению строк текста в буфер, команду I нужно предварительно *ввести*:

```
I↵
```

После этого EDIT будет воспринимать все, что вы введете с клавиатуры (включая возврат каретки, когда вы захотите перейти на новую строку), как текст для вставки, пока вы не нажмете клавишу LINE FEED, обозначаемую здесь символом ↓. Таким образом, в редакторе EDIT ваш сеанс может выглядеть так:

```
*I
THE QUICK
BROWN FOX
```


После того как редактор выдал приглашение *, вы набираете на клавиатуре

I←THE QUICK←BROWN FOX←↓

Все это поступает в буфер, кроме ↓, что просто означает конец вводимого текста. Редактор выдаст новое приглашение *, показывая, что он готов принять следующие команды.

Теперь буфер содержит все, что мы собирались записать, и нам надо передать его содержимое в выходной файл. Содержимое буфера пересылает в выходной файл команда **EX**. Кроме того, она *закрывает* выходной файл и отыскивает для него место на диске в зоне данного пользователя. В редакторе **TECO** **EX** помимо этого возвращает вас в режим команд монитора. В редакторе **EDIT** в результате ввода **EX** появляется приглашение # на ввод новых команд открытия файла. Чтобы передать управление от этого редактора к монитору, нужно нажать **^Z** (вспомним, что это **CONTROL-Z**); так устроено большинство системных программ фирмы DEC.

Снова оказавшись в режиме команд монитора, введите **DIR** ←|, и вы получите *справочник* своих файлов. Можно убедиться, что **TEST.TXT** существует! Для того чтобы проверить содержимое файла, обратитесь к монитору с помощью команды

TYPE TEST.TXT←

и вы все увидите.

УПРАЖНЕНИЕ. Освойте запись файлов, содержащих различные тексты. Определите для себя результат действия **RUBOUT**, **BACKSPACE** и **^U** при работе с ошибками. Что произойдет, если вы нажмете **^Z** без предварительного ввода команды **EX**? Каков эффект от использования **^Z**? Что произойдет, если имя, которое вы даете выходному файлу, принадлежит уже существующему файлу?

Редактирование файла. Предположим, вы хотите исправить то, что уже записали в свой файл **TEST.TXT**. Сначала повторите описанную выше процедуру до того места, когда редактор готов принять команды открытия файла. Работа с текстом может происходить только в буфере, поэтому мы должны уметь пересылать данные из нашего файла в буфер; т. е. **TEST.TXT** должен быть открыт для *ввода*. Исправленные данные поступают обратно в **TEST.TXT** и образуют обновленную версию этого файла, поэтому **TEST.TXT** должен быть открыт и для вывода. В **EDIT** в

ответ на приглашение редактора `#` для этого нужно набрать

`TEST.TXT<TEST.TXT`

и ввести эту команду с помощью `←|`. Здесь **TEST.TXT** сначала упомянут как выходной файл, а после литеры `<`, которая требуется по синтаксису данной команды, — как входной файл. Поэтому, если вы хотите, чтобы новая версия вашего файла имела новое имя, скажем **TEST1.TXT**, то в ответ на приглашение `#` введите другую команду:

`TEST1.TXT<TEST.TXT`

Какой бы редактор вы ни использовали, важно представлять себе, что процесс редактирования состоит из трех этапов. Текст поступает из входного файла в буфер редактирования, обрабатывается в буфере и посылается из буфера редактирования в выходной файл. Очень часто один и тот же файл становится как входным, так и выходным: для такого случая, как правило, предусмотрены специальные команды. Например, в ряде систем с редактором **EDIT** можно ввести такую команду в режиме команд *монитора*:

`EDIT TEST.TXT`

В результате мы не только открываем **TEST.TXT** для ввода и вывода, но и считываем текст из данного файла в буфер. Кроме того, после завершения сеанса редактирования команда **EX** снова передаст управление непосредственно монитору.

Однако если никакая особая команда редактирования не была использована, то наш файл открыт для ввода и вывода, а буфер пуст. Мы лишь образовали двунаправленный канал для взаимодействия между нашим файлом и памятью, а само взаимодействие еще не осуществилось. Нужно ввести команду считывания текста в буфер. В редакторе **EDIT**, например, это команда **R**.

Считав текст в буфер, мы можем проверить, то ли это, что нам требуется, с помощью команды распечатки текста на терминале. В редакторе **EDIT** это команда **L**. После ввода команды **L** редактор напечатает

`THE QUICK`

т. е. мы получим первую *строку* своего файла. Заметим, что символ `←|`, который редактор воспринимает как ограничитель строки, рассматривается как часть этой строки. Чтобы получить

распечатку двух строк нашего файла, нужна команда **2L** и т. д. Поскольку наш файл содержит только две строки текста, то **100L** приведет к тому же результату, что и **2L**.

Команда **L** печатает строку текста, начиная с редакторского индикатора позиции, или *указателя*. Поскольку исправления текста могут производиться только по месту расположения указателя, важно научиться по желанию передвигать его. В начале сеанса редактирования после считывания текста редактор помещает указатель в начало буфера. Введите команду перемещения указателя на одну строку; в редакторе **EDIT** это команда **A**. Затем команда **L** или **2L** выдаст

BROWN FOX

С помощью команды **—L** можно получить строку, непосредственно предшествующую указателю. Заметьте, что эта команда не передвигает указатель; для перемещения указателя нужна специальная команда — в данном случае **A**. В этих командах можно использовать любое целое число — положительное или отрицательное.

УПРАЖНЕНИЯ. 1. Что произойдет в вашем редакторе, если ввести команду перемещения указателя на число строк, превосходящее число строк текста в буфере?

2. Замените **TEST.TXT** на файл с тем же именем, содержащий текст ¹⁾

THE QUICK BROWN FOX JUMPS
OVER THE LAZY DOG

3. Используя команду **I** (или ее эквивалент в вашем редакторе) для вставки текста по месту расположения указателя, преобразуйте **TEST.TXT** к виду

THE QUICK BROWN FOX JUMPS
AGAIN AND AGAIN
OVER THE LAZY DOG

Можно ли так же просто использовать команду **EX**?

4. Каково, по-вашему, назначение «вспомогательного» файла **TEST.BAK**, который вы обнаружите в своем справочнике?

5. Что произойдет, если вы нажмете **^C** перед тем, как закрыть файл с помощью команды **EX**? Проверьте еще раз содержимое вашего справочника.

¹⁾ Этот текст построен так, что в нем хотя бы раз встречается каждая буква латинского алфавита. В переводе он звучит так: «Быстрый рыжий лис перепрыгивает через ленивого пса». — *Прим. ред.*

6. В редакторе EDIT команда K удаляет всю строку текста. В команде K можно использовать любое целое число — положительное или отрицательное. Преобразуйте свой файл еще раз:

THE QUICK BROWN FOX JUMPS
OVER THE LAZY DOG
AGAIN AND AGAIN

Проверьте содержимое TEST.TXT и TEST.BAK.

7. Снова исправьте свой файл, чтобы там можно было прочитать

THE QUICK BROWN FOX JUMPS
AGAIN AND AGAIN OVER THE LAZY DOG

Команды контекстного поиска. Теперь мы знаем, как формировать строки текста. Имея в своем распоряжении набор команд, мы можем по желанию производить любые изменения в файле, удаляя строки и вставляя новый вариант. Обидно, однако, идти на большие затраты ради незначительных исправлений. Пусть, например, мы хотим добавить точку после DOG в версии файла TEST.TXT в упр. 2. Вставлять точку мы умеем, поскольку можем передвинуть указатель на нужное место. Заметьте, что указатель редактора всегда находится либо между двумя литерами, либо перед первым символом буфера, либо после последнего. Считается, что он *никогда* не указывает на саму литеру. Мы хотим вставить точку после буквы G в слове DOG и перед символом ↵, который идет за этим словом. Туда мы и должны переместить указатель.

Как правило, в редакторах есть команда поиска требуемого текста, которая помещает указатель сразу за этим текстом. В редакторе EDIT это команда G. Синтаксис команды G такой же, что и у команды I. Команда G ищет первое вхождение данного текста, начиная с позиции, в которой находится указатель к моменту ввода команды. В нашем примере при поиске символа не возникает неоднозначности. Поэтому, когда редактор готов принять команды редактирования текста, мы печатаем

G↵G↓

где с помощью ↵ происходит ввод команды G, символ G обозначает разыскиваемый текст, а ↓ (LINE FEED) — ограничитель текста.

Чтобы определить положение указателя, мы можем использовать уже известную нам команду распечатки текста, в данном случае L. Команда L выдаст из буфера текст, расположенный от указателя и до ближайшего символа ↵ включительно. (Что делает команда —L? 0L?)

Отметим, что **K** — команда удаления строки — сотрет текст, напечатанный той командой **L**, перед которой стоит такое же число (положительное, или отрицательное, или нуль). Проверьте, как работает команда перехода на следующую строку (это команда **A**), если указатель находится не в начале строки.

Вам следует попрактиковаться в использовании команды поиска, в данном случае **G**, для перемещения указателя по всем направлениям. Для поиска второго вхождения указанного текста можно использовать команду **2G** и т. д. (Что произойдет, если при вводе команды **G** данного текста нет в буфере или он есть, но находится до указателя? Где будет указатель после безрезультатного поиска? Где находится указатель после того, как вслед за словом **DOG** будет поставлена точка?) Как правило, существует команда возврата указателя в начало буфера; в редакторе **EDIT** это **B**.

Пусть мы хотим заменить **JUMPS** на **JUMPED**. В некоторых редакторах есть специальная команда, в которой сначала нужно указать существующий текст, а потом тот, который требуется. В редакторе **TECO** эта операция даже объединена с поиском: если указатель находится раньше слова **JUMPS**, то нам достаточно напечатать

FSJUMPS\$JUMPED\$\$

При этом **TECO** по команде **FS** ищет текст, стоящий перед первой литерой **\$** (**ESCAPE**), и заменяет его на новый текст, который идет дальше до завершающих литер.

Большинство редакторов, однако, не столь удобно. Обычная процедура состоит в поиске **JUMP**, удалении **S** и вставке **ED**. В **EDIT** командой удаления следующей за указателем литеры является **D**; если перед ней стоит положительное или отрицательное число, то можно удалить несколько литер до или после указателя. При использовании этой команды надо помнить, что пробелы и знаки пунктуации — это тоже литеры. Иногда требуется переместить указатель на несколько литер вперед или назад. В редакторе **EDIT** это делается с помощью команды **J**, перед которой ставится соответствующее положительное или отрицательное число.

УПРАЖНЕНИЯ. 1. Сформируйте файл, содержащий текст ¹⁾

ALL THAT GLISTERS IS NOT GOLD
SHAKESPEARE
1596

и вернитесь в режим команд монитора.

¹⁾ «Не все то золото, что блестит». — Шекспир, 1596. Далее приведены аналогичные высказывания Сервантеса и Миддлтона. — *Прим. перев.*

2. Исправьте файл так, чтобы он содержал

ALL IS NOT GOLD THAT GLISTERS
CERVANTES
1615

и вернитесь в режим команд монитора.

3. Снова исправьте файл, чтобы он стал таким:

ALL IS NOT GOLD THAT GLISTENETH
MIDDLETON
1617

Пусть мы хотим преобразовать первоначальную версию **TEST.TXT** так, чтобы весь текст располагался в одну строку. Нужно найти **QUICK**, удалить символ `↵`, который теперь после успешного поиска стоит за указателем, и вставить пробел. Удаление `↵` не вызовет никаких затруднений, поскольку нам известно, что

нажатие клавиши CARRIAGE RETURN приводит к возврату каретки и переводу строки.

Возврат каретки заключается в том, что курсор терминала возвращается в начало той же строки; если (что маловероятно) только это и требуется, то достаточно нажать `^M`. Перевод строки переводит курсор терминала на одну строку вниз; для этого надо нажать `LINE FEED`, или, что одно и то же, `^J`.

УПРАЖНЕНИЯ. 1. Проверьте самостоятельно, как работают `^J` и `^M` по отдельности, когда печатающая головка или курсор терминала находится не в начале строки.

2. Убедитесь, что в первоначальной версии **TEST.TXT** между словами **QUICK** и **BROWN** на самом деле находятся две литеры. Посмотрите, что произойдет, если вы удалите одну из них.

Страничная организация. Объем текста, считываемый командой **R** редактора **EDIT** из входного файла в буфер, называется *страницей*. В данном случае это слово имеет скорее технический, нежели обычный типографский смысл. Наш файл **TEST.TXT** был настолько мал, что нам не надо было беспокоиться насчет страниц. Однако большие файлы должны быть разделены на страницы по двум причинам: во-первых, не во всех редакторах приветствуются попытки считать в буфер редактирования больше, чем он может вместить; во-вторых, с помощью странично-ориентированных команд работать в большом файле гораздо проще, если он имеет страничную организацию. Как правило, предполагается, что буфер редактирования достаточно велик,

чтобы вместить содержимое экрана дисплея или текст, напечатанный через два интервала на странице обычного формата.

В редакторах фирмы DEC наиболее распространенным указателем конца страницы является символ FORM FEED или эквивалентный ему $\wedge L$. Если нужно перейти на новую страницу в TЕСO, в текстовой строке, вставляемой с помощью команды I, просто печатается $\wedge L$. Однако в редакторе EDIT сначала надо завершить символом \downarrow вставляемый текст, затем ввести команду F (т. е. нажать F_{\leftarrow}) для образования в тексте символа FORM FEED, после этого перейти к следующей команде I.

После того как в странице, находящейся к настоящему моменту в буфере, завершился весь процесс редактирования, ее можно отправить в выходной файл. В EDIT содержимое буфера посылает в выходной файл команда N; кроме того, она очищает буфер и считывает следующую страницу текста в буфер, если во входном файле еще есть текст. В некоторых редакторах для того, чтобы выполнить эти операции, может потребоваться несколько команд.

В ряде редакторов есть команды поиска, которые, не ограничиваясь содержимым буфера, проверяют весь входной файл. При использовании такой команды важно представлять себе, что делает редактор со страницами входного файла, в которых нет искомого текста. В редакторе EDIT команда N последовательно считывает страницы входного файла в буфер, просматривает их и отправляет в выходной файл, если требуемый текст еще не обнаружен. Команда P, однако, не выводит, а просто стирает содержимое буфера перед тем, как считать туда следующую страницу. Удобно иметь в своем распоряжении обе команды, но очень раздражает, когда по ошибке используется не то.

Вспомогательные буферы. Во многих редакторах кроме буфера редактирования есть еще один или несколько буферов, что облегчает пересылку блоков текста и вставку одинакового текста в разные части файла.

Предположим, что создается файл, в котором некоторый блок текста встречается более одного раза. Сначала нужно ввести текст в буфер редактирования с помощью команды I. Затем используется специальная команда копирования этого текста в особый буфер. Как правило, синтаксис этой команды тот же, что и синтаксис команды выдачи текста на терминал. Они действительно очень похожи: одна записывает текст в область памяти, другая — на терминал. В редакторе EDIT есть только один дополнительный буфер, называемый буфером *сохранения*. Команда сохранения строк текста S используется в этом редакторе так же, как и L, с той разницей, что в S при указании нескольких строк текста допускаются лишь положительные числа.

Помните, что текст все еще находится в буфере редактирования и должен быть удален оттуда, если он там больше не нужен. Позднее, чтобы содержимое буфера сохранения вставить в буфер редактирования непосредственно перед указателем, используется еще одна команда; в редакторе EDIT это U. Содержимое буфера сохранения не портится, поэтому этот же текст может быть вставлен еще в каком-нибудь месте.

УПРАЖНЕНИЯ. 1. Образуйте файл, состоящий из трех страниц, в которых последовательно располагаются слова **ВТОРАЯ СТРАНИЦА, ТРЕТЬЯ СТРАНИЦА** и **ПЕРВАЯ СТРАНИЦА**.

2. Не используя команды вставки, исправьте ваш файл так, чтобы текст на каждой странице соответствовал порядку страниц в файле.

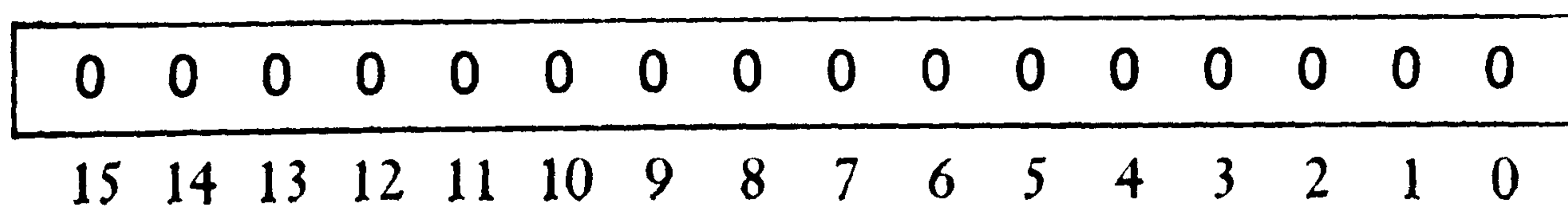
1.3. Запоминание и выборка информации

В § 1.2 мы довольно расплывчато определили программу как последовательность команд для вычислительной машины. При помощи процесса, который мы обсудим ниже, команды перед выполнением должны быть помещены в память ЭВМ.

Память. Память машины состоит из очень большого числа маленьких магнитных или электрических элементов. Долгое время основным элементом был крошечный тороидальный ферромагнетик (*память на ферритовых сердечниках*). Сердечник легко намагничивается в одном или другом направлении в зависимости от направления тока в проводе, проходящем через отверстие тора. Позже стали использовать полупроводниковые элементы (*триггеры*), природа реакции которых на электрические импульсы скорее электрического, нежели магнитного характера. Независимо от физической природы основная идея в обоих случаях одна и та же: память состоит из набора элементов, или *битов*, каждый из которых может находиться в одном из двух *состояний* (для памяти на ферритах два состояния соответствуют двум направлениям намагничивания). Электросхема позволяет другим частям вычислительной системы узнавать состояние каждого бита (*считывать* из памяти) или устанавливать любой бит в нужное состояние (*записывать* в память). Удобной и краткой формой записи этих состояний, напоминающей о вычислительной функции ЭВМ, служат 0 и 1. Нас не интересует, какому направлению намагничивания (по часовой стрелке или против) соответствует 1,— это забота инженеров. Заметим, однако, что, хотя не важно, какому состоянию приписывается 0, а какому 1, вся остальная аппаратура должна отвечать принятому соглашению. Когда нам придется заниматься арифметикой, то надо, чтобы все эти нули и единицы верно изображали числа. Схема,

выполняющая сложение, должна функционировать так, чтобы $0+0$ оставалось равным 0, но $1+1$ не оказалось равно 1. Говорят, что бит, содержащий 1, *установлен*, а бит, содержащий 0, *сброшен*.

Не составляет особого труда закодировать информацию с помощью последовательности битов, принимающих два состояния, что подтверждается существованием азбуки Морзе. Однако в один бит много информации не запишешь. Поэтому разработчики вычислительных машин организуют память так, чтобы ее основная компонента, или *слово*, представляла собой группу битов, достаточную для хранения солидной порции информации. Каждое слово ЭВМ PDP-11 состоит из 16 битов. Состояние слова можно показать на диаграмме.



Обратите внимание, что биты нумеруются от 0 до 15 справа налево.

Двоичная и восьмеричная системы счисления. Как мы убедились, состояние битов в слове памяти ЭВМ PDP-11 (*содержимое слова*) можно изобразить в виде последовательности из шестнадцати нулей и единиц. Числовые данные, текст вроде того, который мы записывали в файл в § 1.2, — все это примеры информации, которую нужно уметь кодировать в виде содержимого слов памяти.

При кодировании чисел последовательные биты изображают последовательно идущие разряды в позиционной системе счисления. По существу, это обычный способ представления чисел. Нам известно, что 1000 в десять раз больше, чем 100, так как в 1000 единица расположена на одну позицию левее, чем в 100. *Десятичной* системой счисления (десять — *основание* системы счисления) мы можем пользоваться, имея в своем распоряжении символы (*цифры*) для представления чисел, меньших десяти. Только после того, как в каком-то разряде мы достигли 9, для получения следующего числа нужно сдвинуться на разряд влево.

Поскольку в слове машины есть лишь нули и единицы, то при появлении 1 в некотором бите для получения следующего числа мы должны передвинуться на бит влево. Таким образом, мы представляем числа в *двоичной* системе счисления, где в качестве основания системы вместо десяти выступает двойка. В такой системе вместо идущих справа налево разрядов единиц, десятков, сотен, тысяч и т. д. будут стоять разряды единиц, двоек, четверок, восьмерок, «шестнадцаток» и т. д. Слово «бит» факти-

чески есть сокращение словосочетания **binary digit** (двоичная цифра).

Например, так как число девятнадцать равно шестнадцать плюс два плюс один, то оно изображается как **10 011**. Как и в случае десятичного представления, мы группируем цифры по тройкам для простоты восприятия. Кратко это записывается так:

$$D\ 19 = B\ 10\ 011$$

где **D** обозначает десятичную, а **B** — двоичную (**binary**) систему счисления. В двоичном представлении левая единица стоит в разряде «шестнадцаток», по нулю в разрядах восьмерок и четверок и по единице в разрядах двоек и единиц. Итак, число девятнадцать можно записать в слово памяти PDP-11 следующим образом:

0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1
15			12					9				6			3	0

Заметим, что порядковый номер бита дает соответствующую степень двойки, которая обозначается цифрой 1 в этом бите; т. е. мы сразу же можем узнать десятичное представление содержимого слова: $2^4 + 2^1 + 2^0 = 16 + 2 + 1 = 19$.

С использованием двоичной системы счисления связаны некоторые неудобства, поскольку человеку трудно воспринимать даже небольшие двоичные числа по причине их громоздкости. Например, не только утомительно вычислять десятичный эквивалент чисел **10 010 110 101** и **10 010 101 101**, но даже не сразу определишь, что они различны. Их десятичное представление гораздо компактнее: 1205 и 1197.

УПРАЖНЕНИЕ. Проверьте эквивалентность десятичного и двоичного представлений этих чисел.

Перевод чисел из двоичной системы счисления в десятичную и обратно — скучное занятие; это работа для машины, а не для человека. Однако, чтобы в полной мере использовать возможности ЭВМ, нам нужно научиться легко интерпретировать содержимое слова машины. К счастью, существует своего рода компромисс между десятичной и двоичной системами счисления — это *восьмеричная* система счисления, основание которой *восемь*. Числа в восьмеричной системе счисления легко читаются; кроме того, очень прост переход от двоичного представления к восьмеричному.

Давайте сначала посмотрим, почему же этот переход так прост. В восьмеричной системе в нашем распоряжении есть

цифры 0, 1, 2, 3, 4, 5, 6, 7; после семерки для увеличения числа мы должны передвинуться на один разряд влево. Таким образом, в восьмеричной системе счисления 10 изображает число восемь

$$O\ 10 = D\ 8$$

Так как в десятичной системе счисления $8=2^3$, то сдвиг на один разряд влево в восьмеричной системе счисления соответствует сдвигу на *три* разряда влево в двоичной системе. Иначе говоря, преобразование из двоичной системы в восьмеричную осуществляется путем замены справа налево каждой *триады* двоичных цифр на одну восьмеричную цифру. Замена всегда возможна, поскольку триадой двоичных цифр можно представить любое число от нуля до семи (т. е. все восьмеричные цифры).

$B\ 000 = O\ 0$	$B\ 100 = O\ 4$
$B\ 001 = O\ 1$	$B\ 101 = O\ 5$
$B\ 010 = O\ 2$	$B\ 110 = O\ 6$
$B\ 011 = O\ 3$	$B\ 111 = O\ 7$

Например, $B\ 10\ 011 = O\ 23$. Мы уже выяснили, что это равно $D\ 19$, кроме того, это следует из восьмеричного представления числа: дважды восемь плюс три есть девятнадцать.

Рассмотрим теперь число $D\ 1205 = B\ 10\ 010\ 110\ 101$. Двоичные триады справа налево таковы: $O\ 5$, $O\ 6$, $O\ 2$, $O\ 2$. То есть восьмеричное представление этого числа есть $O\ 2265$; что означает $2 \times \times (\text{восемь} \times \text{восемь} \times \text{восемь}) + 2 \times (\text{восемь} \times \text{восемь}) + 6 \times (\text{восемь}) + 5$.

Чтобы преобразовать восьмеричное представление числа в двоичное, выполним процесс в обратном порядке. Например, мы имеем $O\ 734$; тогда $O\ 4 = B\ 100$; $O\ 3 = B\ 011$; $O\ 7 = B\ 111$. Итак, $O\ 734 = B\ 111\ 011\ 100$. В десятичной системе счисления это есть $7 \times (8 \times 8) + 3 \times (8) + 4 = D\ 476$.

Важно запомнить соотношения

$$\begin{aligned} D\ 10 &= O\ 12 \\ D\ 8 &= O\ 10 \\ D\ 64 &= O\ 100 \end{aligned}$$

УПРАЖНЕНИЯ. 1. Почему столь просты преобразования между системами счисления с основаниями два и восемь?

2. Равнозначны ли по сложности следующие преобразования: а) от основания три к основанию двенадцать; б) от основания три к основанию девять; в) от основания два к основанию шесть; г) от основания два к основанию четыре?

Опишите, как осуществляются преобразования в тех случаях, когда они просты.

3. Что вы скажете, если вас попросят перевести число 59 из семеричной системы счисления в десятичную?
4. Переведите в десятичную систему счисления: а) $O\ 37$; б) $O\ 40$; в) $B\ 1\ 111$; г) $B\ 11\ 110$.
5. Переведите в восьмеричную систему счисления: а) $D\ 37$; б) $D\ 40$; в) $B\ 1\ 111$; г) $B\ 11\ 110$.
6. Переведите в двоичную систему счисления: а) $D\ 37$; б) $O\ 37$; в) $D\ 32$; г) $O\ -32$.
7. Чему равно $O\ 100-1$: а) в восьмеричной системе счисления; б) в десятичной системе счисления?
8. Положительна или отрицательна разность $O\ 15-O\ 60$? Почему?

Преобразования из двоичной системы в восьмеричную и обратно осуществляются на ЭВМ довольно быстро, так что обычно системные программы обмениваются с пользователем числовой информацией в восьмеричной системе счисления. Поэтому необходимо приобрести некоторый навык в использовании восьмеричной системы. Мы не имеем в виду выполнение сложных восьмеричных вычислений или преобразований между восьмеричными и десятичными числами; в свое время мы узнаем, как научить это делать вычислительную машину. Нужно только привыкнуть к счету в восьмеричной системе счисления. Вскоре уже не покажется странным, что в восьмеричной системе 17 плюс 762 равно 1001 , а разность 762 и 17 есть 743 (проверьте!). Изредка нам может понадобиться грубая оценка числа, представленного в восьмеричной системе счисления. В данном примере, так как $O\ 1000 = D\ 8 \times 8 \times 8 = D\ 512$, можно сказать, что число $O\ 762$ находится где-то около пятисот; более точная информация потребуется нечасто.

Код ASCII. Мы должны уметь кодировать в машинных словах не только цифры, но и буквы; кроме того, все символы с клавиатуры терминала. Известен код, который каждому символу ставит в соответствие число. Этот код, широко применяемый во многих разных машинах, называется American Standard Code for Information Interchange (Американский стандартный код для обмена информацией). Обычно используется аббревиатура ASCII (произносится «эз-ки»). Полный стандартный терминал содержит 127 различных символов (это десятичное 127). Сюда входят не только заглавные и строчные буквы, цифры и специальные символы, но и их комбинации типа управляющих символов, каждый из которых в коде ASCII рассматривается как один символ. Например, $\wedge A$ в коде ASCII есть 1. Предположим, нам как-то удалось образовать в машине слово, содержащее число 1. Это значит, что при чтении справа налево первый бит в слове

содержит 1, остальные — нули. В зависимости от выполняемых действий может потребоваться, чтобы эта единица означала либо ^A, либо число 1. Поэтому с самого начала надо представлять себе, что ЭВМ не может «знать», что мы имеем в виду, пока мы соответствующим образом не проинструктируем ее.

Символы терминала интерпретируются в коде ASCII как восьмеричные цифры от 0 до 0 177. Заметим, что D 128 есть $2 \times \times$ (восемь \times восемь), т. е. равно 0 200. Вычитая из этого числа единицу, получаем $D\ 127 = 0\ 177$ — количество различных символов в коде ASCII.

Важно понять, почему $0\ 200 - 1 = 0\ 177$. Рассмотрим, что получится в результате сложения 1 и 0 7. В разряде единиц их число увеличится до восьми, поэтому мы должны сделать перенос влево, в разряд восьмерок: $0\ 7 + 1 = 0\ 10$. Аналогично $0\ 17 + 1 = 0\ 20$. При сложении 1 с 0 77 перенос единицы в разряд восьмерок увеличивает их число до 8, поэтому нужно сделать еще один перенос влево, в разряд (восемь \times восемь): $0\ 77 + 1 = 0\ 100$. Аналогично $0\ 177 + 1 = 0\ 200$ и т. д.

Естественно, символы в коде ASCII можно преобразовать в десятичное представление, однако в этом нет необходимости. Гораздо лучше привыкнуть к этим числам в восьмеричной форме, в которой они обычно и обозначаются. Только нужно помнить, что все цифры не превосходят 7; и поэтому после прибавления 1 в разряд, где уже есть 7, там останется 0, а 1 переносится влево.

Буквы от A до G в коде ASCII:

A	101	E	105
B	102	F	106
C	103	G	107
D	104		

Каков, по-вашему, код ASCII буквы H? Вы, конечно, догадались, что он равен коду G плюс 1; это будет

H 110

и т. д. до

W	127
X	130
Y	131
Z	132

Цифры на терминале имеют такие коды:

0	60
1	61

и т. д. до

7	67
8	70
9	71

Очень удобно, что коды ASCII для последовательно идущих цифр тоже представляют собой последовательные числа; заметим, что можно получить одни из других, добавляя или вычитая 0 60.

УПРАЖНЕНИЯ. 1. Сколько битов требуется для изображения любого символа в коде ASCII?

2. Пусть мы хотим поместить в слова ЭВМ PDP-11 текст в коде ASCII, который может быть напечатан либо прямым шрифтом, либо курсивом (в сложных терминалах есть несколько типов шрифтов). Предложите экономный способ хранения символов вместе с информацией о виде шрифта.

3. Какое максимальное число в двоичной системе счисления можно разместить в шестнадцати битах слова памяти ЭВМ PDP-11?

Адресация. Возможность хранения информации в памяти ЭВМ ничего бы не стоила, если бы мы не могли по необходимости извлекать ее оттуда. Ясно, что нужно каким-то образом отличать одно слово памяти от другого; проще всего их перенумеровать. Номер слова называется его *адресом*.

Теперь с каждым элементом памяти связаны два числа: адрес элемента и его содержимое, выраженное в виде числа в двоичной системе счисления, что соответствует комбинации битов. Эти два числа настолько различаются по смыслу, что беспокойство о возможности их спутать может показаться излишним. На практике, однако, это вполне реально. Пусть, например, мы хотим записать букву В в коде ASCII в элемент памяти с номером 2. Для этого подходит такая команда на языке ассемблера PDP-11:

MOV #102,2

Однако даже опытный программист может по невнимательности написать

MOV 102,2

в результате чего содержимое элемента памяти с номером 2 станет равно содержимому элемента с номером 0 102. С точки зрения машины команда правильная; просто она не выполнит то, что имел в виду программист.

Адрес элемента памяти может быть выражен в двоичной системе счисления и записан в любой элемент памяти в виде комбинации двоичных разрядов. Другими словами, *содержимое одного слова можно рассматривать как адрес другого*. Говорят, что первое слово указывает на второе. В языке ассемблера при наличии такой ситуации ставится символ @. Иначе говоря, команда

MOV @102,2

заменяет содержимое элемента с адресом 2 содержимым элемента, на который *указывает* элемент памяти с номером 0 102. Далее мы обсудим более подробно этот и другие типы адресации (отчего они станут менее загадочными).

Память ЭВМ PDP-11 организована несколько иначе, чем в большинстве машин. В PDP-11 не только любое слово имеет собственный адрес, но и каждая половина слова. Всякое шестнадцатиразрядное слово разбито на два *байта* по восемь битов каждый. Этим достигается простота адресации как всего слова, так и любой его половины, возможно, ценой ошибок.

Два байта каждого слова имеют последовательные адреса. Адрес байта, состоящего из разрядов с номерами от 0 до 7 (*младшего байта*), — всегда четное число; *старший байт* (разряды от 8 до 15) слова имеет следующий по порядку адрес. Таким образом, память можно схематично изобразить так:

Адреса	Байт		Адреса
1	старший	младший	0
3	"	"	2
5	"	"	4
7	"	"	6
11	"	"	10
...

Обратите внимание, что *адреса памяти выражаются в восьмеричной системе счисления*.

Адрес слова памяти совпадает с адресом его младшего байта. Поэтому в PDP-11

адрес слова — всегда четное число.

Важно помнить, что адрес следующего слова получается из адреса предыдущего путем прибавления *двойки*. Может показаться, что совпадение адресов у слова и его младшего байта приводит к недоразумениям, но мы увидим, что это не так.

Центральный процессор. Пусть некоторое слово памяти содержит такую комбинацию двоичных разрядов: **0 000 000 001 000 010**. Вспомним, что это всего лишь запись состояния группы электрических или магнитных элементов. Мы можем приписать этой комбинации двоичных разрядов некий смысл; однако машина не имеет возможности предвидеть ход мыслей программиста и поэтому будет ожидать специальной команды. Нам уже известно, что указанная комбинация разрядов может изображать либо число **0 102**, либо ячейку памяти с адресом **0 102**, либо букву **В** в коде ASCII. Иногда желаемый смысл понятен из контекста. Если машина получила указание переслать эту комбинацию битов в такое устройство терминала, которое образует электрические сигналы для передачи в печатающий механизм (*буфер печатающего устройства терминала*), то схема терминала автоматически напечатает букву **В**.

Та часть машины, которая выполняет команды, называется *центральным процессором* (ЦП). Каждая операция (например, сложение или вычитание), которую может выполнить ЦП, связана с определенным блоком схемы. ЦП должен выбрать подходящий блок и выполнить соответствующую операцию, руководствуясь состоянием двоичных разрядов *регистра команд*. Если левые шесть битов равны **000110**, то ЦП включает блок сложения; если они равны **001110**, то выбирается вычитание. Нам нет необходимости углубляться в то, как физически реализуется этот процесс.

Регистр команд ЦП состоит из шестнадцати битов. Состояние этих разрядов служит ЦП указанием о том, какую операцию нужно выполнить, где найти данные для команды и где хранить результат. Заметьте, что длина регистра команд та же, что и слова памяти. Это отнюдь не совпадение, а фундаментальный принцип организации вычислительной машины.

Машинные команды хранятся в памяти ЭВМ.

Можно, разумеется, представить хранящуюся в слове памяти команду как число в двоичной и восьмеричной системах счисления — точно так же, как и данные. Мы уже затрагивали вопрос о том, каким образом машина «распознает», что представляет собой содержимое слова памяти (**0 102** — это данные, адрес или буква **В**?). К этому надо добавить, видимо, более фун-

даментальный вопрос: откуда компьютер «знает», что слово содержит данные, а не команду?

Простой ответ на этот вопрос состоит в том, что ЭВМ ничего не «знает»! Для нее слова памяти — это просто комбинации двоичных разрядов. *Вопрос о том, рассматривается содержимое конкретного слова как данные или как команда, должен решаться самим программистом.*

С помощью регистра, называемого *счетчиком команд* ¹⁾ (РС), ЦП определяет, какое слово памяти будет выступать в качестве следующей команды. Регистр, как и слово памяти, состоит из шестнадцати битов, но находится не в памяти, а в ЦП и предназначен для выполнения разнообразных действий.

ЦП непрерывно выполняет следующий цикл операций.

1. В регистр команд считывается содержимое того слова памяти, на которое указывает РС (слово при этом не портится).

2. Если в команде, находящейся в регистре команд, сообщается, что данные должны быть выбраны из памяти, то

а) содержимое РС увеличивается на два, чтобы он указывал на следующее слово;

б) в соответствующий регистр данных считывается содержимое слова памяти (его адрес зависит от синтаксиса команды и содержимого слова, на которое теперь указывает РС);

в) если не все необходимые данные получены, то возврат к п. а); в противном случае — переход к следующему шагу.

3. Содержимое РС увеличивается на два.

4. Выполнение команды.

5. Переход к шагу 1.

По существу, аппаратура ЦП запрограммирована на выполнение цикла, состоящего из выборки команды, на которую указывает РС, изменения РС и выполнение самой команды. В некоторых командах ЦП должен выбирать операнды из памяти (как, например, в команде сложения содержимого двух слов памяти); тогда выполняется внутренний «цикл» шага 2.

Как уже отмечалось, в задачу программиста входит не допустить, чтобы в момент готовности принять данные ЦП натолкнулся на слово, содержащее команду, и наоборот. Эта функция возложена и на системные программы. Например, программист может просто написать

INC

102

чтобы увеличить на единицу содержимое элемента памяти с адресом 0 102. Специальная системная программа, называемая

¹⁾ «Program counter», — Прим. перев.

ассемблером, транслирует это в два командных слова

Увеличить элемент памяти
102

Команда увеличения элемента памяти заставляет ЦП один раз выполнить внутренний цикл шага 2. На этом этапе РС содержит адрес второго слова данной последовательности и содержимое элемента памяти 102 будет считано в регистр данных. На шаге 4 содержимое регистра данных увеличится на единицу и затем снова будет отправлено по адресу 102.

УПРАЖНЕНИЯ. 1. Пусть команда **INC 102** хранится в памяти, начиная с адреса 100. Что произойдет, когда выполнится эта команда? Что будет, если она находится в цикле программы и должна выполняться несколько раз?

2. Команда **INC PC** транслируется в одно слово памяти. Можно ли с помощью последовательности

INC	PC
INC	PC

увеличить содержимое РС на два?

3. Команда **ADD #2,PC** увеличит на два содержимое РС. Она занимает два слова памяти

Добавить число к РС
2

Можно ли с помощью последовательности

ADD	#2,PC
ADD	#2,PC

увеличить содержимое РС на четыре?

4. Каков результат выполнения последовательности

ADD	#2,PC
INC	PC

5. Команда вычитания **SUB** транслируется аналогично **ADD**. Каков результат выполнения команды **SUB #4,PC** в программе?

1.4. Выполнение программы

В § 1.1 мы обсуждали круг проблем, связанных с большим разнообразием использующихся в ЭВМ PDP-11 операционных систем. В данном параграфе эти вопросы будут играть большую роль. Может потребоваться все ваше терпение, чтобы разобраться в том, как пропустить весьма простые программы, здесь описанные, в вашей конкретной системе PDP-11.

Независимо от сложности программы мы можем ничего не узнать о ее работе, пока не получим результаты в некоторой осязаемой форме. Другими словами, программа должна представить нам *выходные данные*, чтобы мы могли знать о происходящем. Выходные данные могут быть, например, в виде управляющих сигналов, которые руководят действиями сложной аппаратуры, либо, как мы и будем далее предполагать, в виде напечатанной на терминале информации.

Наша первая программа дает ЭВМ команду напечатать на терминале букву **В** и затем остановиться. В изолированной системе PDP-11 пользователь имеет прямой доступ в буфер печатающего устройства терминала и может поместить туда **O 102**; в результате символ, код ASCII которого есть **O 102** (т. е. **В**), напечатается на терминале. Однако в системе разделения времени нельзя позволить каждому пользователю осуществлять прямое управление электронными каналами, по которым идет связь пользовательских терминалов с машиной. Следовательно, в системе разделения времени ввод и вывод полностью находятся под управлением монитора, кроме случая особо привилегированных пользователей. В такой системе программа монитора инициирует процедуру проверки команд, поступающих в ЦП. Если ваша программа выдаст команду пересылки данных в буфер печатающего устройства вашего терминала, то вмешается монитор, остановит программу и на вашем терминале напечатает сообщение об обращении к неверному адресу.

Операционная система в изолированных машинах PDP-11 также может иметь управляемые монитором ввод и вывод либо в дополнение к возможности организовать его самим пользователем, либо вместо этой возможности. Подпрограммы монитора эффективны и просты в обращении и, как правило, экономят время при программировании. В этом параграфе мы уделим особое внимание подпрограммам монитора, отложив обсуждение управляемого пользователем ввода-вывода до § 4.1.

Вывод, управляемый монитором. Рассмотрим те подпрограммы ввода-вывода, управляемые монитором, которые есть в операционной системе RT-11. Это широко применяемый набор подпрограмм, поскольку их можно реализовать и в других опера-

ционных системах. Вам нужно проверить, есть ли в вашей системе эти подпрограммы, а если нет, то выяснить, что их заменяет.

Литеру в коде ASCII можно послать на терминал, обратившись к монитору с помощью команды

`.TTYOUT`

Обратите внимание, что первой литерой этой команды является точка. Поскольку `TTY` — обычная мнемоника для терминала вычислительной машины, то и мнемоника самой команды `.TTYOUT` вполне разумна.

Нужно сообщить монитору о том, какую информацию ему предстоит направить в буфер печатающего устройства терминала. Мы хотим послать код ASCII буквы **B**. Однако нельзя написать

`.TTYOUT 102`

так как при этом будут выведены семь правых разрядов ячейки памяти с адресом **O 102** в виде литеры в коде ASCII. Правильной будет команда

`.TTYOUT #102`

Уже встречавшийся нам символ `#` указывает, что за ним следуют сами данные (в восьмеричной системе счисления), а не адрес данных.

Одна эта команда выполняет все, что нам требуется. Но, к сожалению, ее недостаточно, чтобы создать файл (назовем его **TEST.MAC**), содержащий лишь одну такую строку. Чтобы понять, чего не хватает, изучим процесс, посредством которого выполняется программа.

Ассемблер. В § 1.3 мы упоминали, что ЦП получает команды в виде групп из шестнадцати двоичных разрядов. Например, команда прибавления 1 к содержимому **PC** выглядит так:

`0 000 101 010 000 111`

Запись этой команды и в восьмеричной системе счисления как **005 207** (проверьте!) совершенно неудобна для программиста. Однако, как уже отмечалось, вместо этого программист может написать команду

INC

PC

которую легко понять или вспомнить, даже если она не делает то, что надо (почему?). Преобразование из этой формы в двоичный код, приемлемый для ЦП, осуществляет *ассемблер*. Ассемблер — это программа, которая имеет в памяти таблицу команд языка ассемблера типа упомянутой только что команды INC. Каждой строке таблицы сопоставлен соответствующий двоичный код. Кроме того, в таблице должно быть определено, как кодировать *операнд* (*операнды*) ¹, в данном случае РС. Заметьте, что эти действия ассемблера не влекут за собой никакого расширения программы; ассемблер лишь *транслирует* ее из одной формы в другую. Отсюда следует, что в распоряжении человека, программирующего на языке ассемблера, находится вся система команд ЭВМ, и писать непосредственно в машинных кодах нет нужды. Однако во многих случаях полезно знать машинный код команды.

Ясно, что язык ассемблера для некой конкретной ЭВМ не есть нечто совершенно уникальное. И в самом деле, для некоторых машин создано несколько языков ассемблера, но отсюда не следует, что это преимущество. Поскольку в математическом обеспечении фирмы DEC для ЭВМ PDP-11 есть очень богатый и выразительный язык ассемблера, то вряд ли кому-то придет в голову создавать другой, разве что в качестве упражнения по программированию. Кроме того, как мы позже убедимся, данный язык ассемблера позволяет программисту образовывать новые команды, которые отвечают его требованиям. Такие придуманные программистом команды называют *макрокомандами*, и название MACRO-11 языка ассемблера PDP-11 отражает эту возможность. Некоторые операционные системы, однако, располагают лишь подмножеством языка MACRO-11 без макросредств; это подмножество называется PAL-11 (Program Assembly Language).

Начальный адрес. Ассемблер не ограничивается переводом вашей программы в машинный код. Он также подготавливает почву для системной программы *загрузчик*, функция которой состоит в загрузке в память машинного кода, содержащего вашу программу, после чего ее можно выполнять. Файл, который ассемблер образует для хранения программы, переведенной в двоичный код, содержит среди прочего информацию о том месте программы, с которого начнется ее выполнение. Эта точка называется *начальным адресом* программы, хотя, конечно, обозначенная таким образом команда еще не имеет адреса, пока загрузчик не поместит программу в память.

¹ Операнды — это объекты, над которыми выполняется операция.—
Прим. перев.

Может показаться, что определение начального адреса излишне. Было бы разумно ожидать, что ЦП начнет выполнять вашу программу с самого начала, дойдет до конца, затем остановится. Однако в этом случае мы лишились бы одного из наиболее ценных качеств сложных программ загрузки — умения во время загрузки компоновать единую программу из независимо оттранслированных подпрограмм. Загрузчик, обладающий таким свойством, называется *компоновщиком*. Все это можно осуществить только тогда, когда какая-то одна из подпрограмм отмечает начало выполнения операций с помощью начального адреса.

Метки. Чтобы отметить какую-то строку в программе, можно использовать *метки*. Программист должен выбирать метки по собственному усмотрению; метки содержат до шести букв или цифр. Сначала всегда стоит буква. Мы хотим дать команде **.TTYOUT** начальный адрес, поэтому для метки подходит слово **START**. Метка должна находиться в начале помеченной строки, затем, без пробела, идет двоеточие:

```
START:      .TTYOUT      #102
```

Между **START** и **.TTYOUT** должны быть пробелы; их количество как здесь, так и в начале строк для ассемблера не играет роли, поэтому удобно использовать табуляцию, чтобы расположить метки и команды в наглядном, легко читаемом формате.

Ассемблер распознает **START** как метку по идущему за ним двоеточию и включает **START** в таблицу меток.

Мы еще не определили адрес строки с меткой **START** как начальный. Эта строка всего лишь помечена словом, напоминающим нам о начальном адресе. Структура языка ассемблера PDP-11 такова, что в последней строке программы должно быть указание ассемблеру о том, какой адрес будет начальным. Однако ассемблер не знает, какая строка последняя, пока ему об этом не сообщат. Иными словами, в *каждой* программе последняя строка должна содержать специальную команду **.END**, сообщающую ассемблеру о необходимости завершить трансляцию. Если за командой **.END** идет выражение, которое использовалось в программе в качестве метки, ассемблер отметит адрес строки с такой меткой как начальный. Итак, наша программа теперь имеет вид

```
START:      .TTYOUT      #102
              .END        START
```


Заметьте, что мы приняли такое расположение программы, при котором команды выравниваются по вертикали.

Типы команд. Обратите внимание, что **.END** — не команда языка ассемблера. Это инструкция *программе ассемблера* о том, что нужно прекратить трансляцию. То есть строка, содержащая **.END**, не транслируется в машинный код (*не генерирует никакой код*). Операторы типа **.END**, хотя и выглядят в программе как команды, предназначенные для ЦП, на самом деле управляют процессом трансляции и называются *псевдооператорами*.

По существу, **.TTYOUT** тоже не является командой языка ассемблера. Это наш первый пример макрокоманды, однако так как она образована не программистом, а операционной системой, то носит название *системной макрокоманды*. Ассемблер транслирует **.TTYOUT** в целую подпрограмму на языке ассемблера. Обсуждение подпрограммы **.TTYOUT** нам придется отложить; заметим только, что она выполняется при помощи монитора.

Ассемблер не будет по собственной инициативе искать файл, где находятся системные макро. Для того чтобы начать поиск каждой используемой системной макро, он должен получить некую команду; такой командой является псевдооператор **.MCALL**. Без этого ассемблер просто будет рассматривать **.TTYOUT** как неопределенный символ. Такое недоверие со стороны программы ассемблера явно неоправданно; будем надеяться, что дальнейшие версии получат соответствующие исправления.

Теперь наша программа имеет вид

	.MCALL	.TTYOUT
START:	.TTYOUT	#102
	.END	START

Эта программа, очевидно, будет правильно транслироваться. Посмотрим, однако, что произойдет, когда программа станет выполняться. ЦП начнет выполнять команды системной подпрограммы, обозначенной **.TTYOUT #102**. По окончании работы подпрограммы будет напечатана буква **B**, и наша программа сделает все, что мы от нее хотели. К сожалению, ничто в программе не указывает ЦП на то, что нужно завершить выполнение программы и передать управление монитору. Это необходимый шаг (почему?), и он осуществляется с помощью системной макрокоманды **.EXIT**. Ассемблер должен быть предупрежден о необходимости искать **.EXIT** в файле системных макро. С этой целью к программе добавляется еще одна строка:

.MCALL .EXIT

Или это можно сделать, как в программе, приведенной ниже. Обратите внимание, что системные макрокоманды и псевдооператоры имеют отличительный признак в виде точки в качестве своего первого символа.

Окончательно полная программа печати буквы В имеет вид

	.MCALL	.TTYOUT,EXIT
START:	.TTYOUT	#102
	.EXIT	
	.END	START

Вам необходимо создать файл **TEST.MAC**, который будет содержать эту программу.

Процесс трансляции. В вашей системе могут быть особые компактные команды трансляции, загрузки и выполнения программ на языке ассемблера. На досуге вы их можете изучить, а мы будем придерживаться более общего пошагового метода.

Первый шаг состоит в выполнении системной программы **MACRO**. Быть может, вы захотите вернуться к обсуждению вопроса о выполнении системных программ в § 1.2. Возможный вид команды таков:

RUN \$MACRO ↵

Однако в системе, которую мы использовали при подготовке этой книги, есть две версии макроассемблера, известные как **\$MACRO** и **\$MACROL**; лишь во второй версии доступен файл системных макрокоманд **.TTYOUT** и **.EXIT**. Мы упомянули об этом, чтобы привести еще один пример тех проблем, с которыми можно столкнуться.

Напечатав символ *****, ассемблер сообщит о своей готовности принять команды. Нам нужно, чтобы ассемблер использовал **TEST.MAC** в качестве входного, или *исходного*, файла и образовал на выходе машинный код (*объектный* файл). Лучше всего назвать файл с машинным кодом таким же именем, но с другим расширением. Укажем для **MACRO** сначала выходной, затем входной файл, отделив их знаком **=**

TEST=TEST ↵

Ассемблер образует файл с двоичным кодом, даст ему стандартное расширение **.OBJ** и поместит в вашу зону на диске. Вы можете убедиться в наличии **TEST.OBJ** в своем справочнике, нажав предварительно **^Z**, чтобы вернуться в режим команд монитора.

В зависимости от особенностей вашей системы всю эту процедуру можно заменить одной командой монитора

COMPILE TEST ↵

Более точно под *компиляцией* понимается процесс получения файла, содержащего двоичный код, из файла, написанного на языке высокого уровня. Однако в некоторых системах возможность использования команды **COMPILE** распространяется и на программы, написанные на языке ассемблера. Заметим, что имени файла не даются расширения при вызове программы ассемблера; макроассемблер воспринимает **TEST** как обозначение исходного файла, если обнаружит **TEST.MAC** в вашей зоне на диске.

Системная макробибблиотека. Может случиться, что ассемблер ответит на только что описанную процедуру сообщением об ошибке из-за наличия в вашей программе неописанных символов. Ими, несмотря на наши предосторожности, окажутся **.TTYOUT** и **.EXIT**. Используемая вами версия макроассемблера, видимо, не совместима с форматом файла, в котором хранятся системные макро (системной *макробибблиотекой*). Выясните это у администратора вашей системы. С другой стороны, системная макробибблиотека может оказаться недоступной для ассемблера. Тогда попросите администратора системы скопировать ее в вашу зону; в большинстве систем она называется **SYSMAC.SML**, но в любом случае она может быть описана как макробибблиотека системы RT-11.

Распечатка файла. Если по поводу программы на языке ассемблера возникают какие-то вопросы, то при их разрешении может помочь изучение кода, образованного ассемблером. Однако этого нельзя сделать с объектным файлом; нет смысла заставлять монитор распечатывать его (почему?).

При использовании соответствующей команды монитор выдаст читаемую версию объектного файла. Она называется *листингом* файла, который можно получить, сообщив ассемблеру о том, что выходных файлов должно быть *два*:

TEST,TEST=TEST ↵

Второй **TEST** интерпретируется как запрос листинга файла с таким же именем; ассемблер образует **TEST.LST** и поместит его в вашу зону на диске.

На рис. 1.1 представлен листинг файла нашей программы **TEST.MAC**. Первая колонка листинга содержит номер строки

в программе. Эта информация выдается независимо от того, генерирует ли данная строка какой-либо код. Во второй колонке даются адреса кодовых строк, которые сами находятся в третьей

```
.MAIN. RT-11 MACRO VM02-11 26-JAN-79 09:28:07 PAGE 1
```

```
1          .MCALL .TTYOUT,.EXIT
2 000000      START: .TTYOUT #102
3 000010          .EXIT
4          000000'      .END      START
```

```
.MAIN. RT-11 MACRO VM02-11 26-JAN-79 09:28:07 PAGE 1+
SYMBOL TABLE
```

```
START      000000R
. ABS.      000000      000
            000012      001
ERRORS DETECTED: 0
FREE CORE: 18899. WORDS
```

Рис. 1.1. Программа вывода.

колонке. Обратите внимание, что они записываются в восьмеричной системе счисления.

Из этого листинга следует, что ячейка 0 является начальным адресом. В адресной колонке, однако, записывается не фактический адрес, по которому происходит загрузка команд, а адрес, отсчитываемый *относительно начального адреса*. Поэтому начальный адрес соответствует нулевому относительному адресу. Определение загрузочных адресов не входит в функцию ассемблера.

Можно также выяснить, какой код генерируют системные макрокоманды типа **.TTYOUT** и **.EXIT**, если заставить ассемблер распечатать макрорасширения, для чего используется псевдооператор

```
.LIST      ME
```

хотя это пока вам мало поможет.

Заметьте, что протокол файла не содержит имени файла. Это можно исправить, включив в программу псевдооператор **.TITLE**. За словом **.TITLE** указывается имя, которое мы хотим получить в листинге; это может быть имя файла, содержащего программу (но не обязательно):

```
.TITLE      TEST
```

Если нет оператора **TITLE**, то ассемблер дает имя **.MAIN**, что видно из рисунка.

УПРАЖНЕНИЯ. 1. Оттранслируйте без ошибок **TEST.MAC** или самый близкий к нему эквивалент в вашей системе.

2. Просмотрите этот параграф еще раз и подумайте, к чему приведет исключение одного или нескольких шагов, о которых мы говорили как о существенных.

3. Имеет ли значение, в каком месте программы находится псевдооператор **.MCALL**; оператор **.LIST**; оператор **.TITLE**?

Компоновщик. Если вы используете операционные системы RSX-11 или IAS, то этот параграф нужно читать вместе с документацией по вашей системе. Он будет ближе к действительности для пользователей систем RT-11 и RSTS.

Как мы уже отмечали, ассемблер не определяет место загрузки вашей программы в память. Он лишь снабжает программу относительными адресами, которые отсчитываются от начального адреса, вместе с указанием на то, что данное число — относительный, а не *абсолютный* адрес. Относительный адрес помечается апострофом «'» в третьей колонке протокола файла. При помощи рис. 1.1 убедитесь, что в обращении к метке **START** в операторе **.END** используется относительный адрес. Это, по существу, единственный код такого рода в нашей программе.

На следующем этапе выполнения программы необходимо получить из **TEST.OBJ** файл на диске, в котором все относительные адреса заменены на абсолютные. Это функция программы **LINK**, и мы обращаемся к монитору с соответствующей командой:

RUN \$LINK↵

Программа **LINK** выдаст символ *, и мы введем спецификации входного и выходного файла подобно тому, как это уже делалось в **MACRO**.

TEST,TEST=TEST

LINK работает с файлами с расширением **.OBJ**, поэтому он выберет в вашей зоне на диске **TEST.OBJ** в качестве входного файла. Первым из двух файлов, упомянутых в качестве выходных, станет **TEST.SAV**. Это файл *образа памяти*, который отражает состояние памяти в тот момент, когда программа будет наконец загружена. В нем установлен начальный адрес (адрес *передачи управления*) и все относительные адреса соответственно *смещены*. Программы монитора, как правило, используют первые 400 (восьмеричное число) слов памяти для управления, поэтому начальный адрес обычно равен 1000 (почему не 400? 1002?).

Второй выходной файл, образованный программой LINK, содержит операции в доступном для чтения виде. Он называется *картой загрузки* и имеет расширение **.MAP**. Распечатайте файл **TEST.MAP** и изучите его.

Осталось только загрузить файл образа памяти в память и начать выполнение с начального адреса. Это делается с помощью одной команды монитора

RUN TEST↵

после чего символ **B** появится на терминале. В системе RT-11 мы можем разбить действие команды **RUN** на такие составляющие:

GET TEST↵

что производит загрузку программы в память, и

START↵

что помещает начальный адрес в **PC**, начиная таким образом выполнение программы.

УПРАЖНЕНИЯ. 1. Выполните программу **TEST.MAC** или ее эквивалент применительно к вашей системе.

2. Напишите программу, которая печатает сообщение **ПРОВЕРОЧНЫЙ ВЫВОД**, за которым следует символ ↵.

3. Что будет, если не включить в программу **.EXIT**?

2. ОСНОВЫ

2.1. Регистры

Мы описали регистры как аналоги ячеек памяти, но расположенные внутри ЦП и предназначенные для выполнения специальных функций. Число регистров зависит от модели ЭВМ PDP-11, что, однако, скрыто от рядового программиста с помощью мониторов. Пользователю всегда доступны восемь регистров. В системе разделения времени монитор постоянно передает управление от одной программы к другой. Это не мешает каждой программе использовать одни и те же регистры, так как первое, что делает программа монитора, когда отбирает ЦП у программы пользователя, это запоминает в основной памяти содержимое регистров с последующим их восстановлением при возврате управления.

Чтобы проиллюстрировать использование регистров, изменим программу из § 1.4. Поместим данные в коде ASCII, которые мы хотим вывести, в первый регистр (регистр 0) и затем выдадим содержимое регистра 0 на терминал.

Ассемблер распознает регистр по символу %. Таким образом, регистр 0 в программе обозначается %0; его содержимое можно вывести на терминал с помощью команды

```
.TTYOUT    %0
```

Первые шесть регистров, от 0 до 5, стало общепринятым обозначать R0÷R5. (Регистры 6 и 7 являются регистрами специального назначения и обозначаются особым образом.) Этого соглашения нужно придерживаться, чтобы облегчить общение между пользователями PDP-11. К сожалению, многие версии ассемблера не распознают эти имена. Поэтому иногда необходимо объявить ассемблеру, что R0 соответствует %0 и т. д. Это достигается включением в программу *оператора присваивания* R0==%0 (для других используемых в программе регистров аналогично). Перед знаком = не должно быть пробела:

```
R0=%0
```

```
...
```

```
...
```

```
.TTYOUT    R0
```

Однако, прежде чем вывести содержимое R0, в регистр нужно поместить соответствующие данные. Командой занесения данных в элемент памяти (регистр или ячейку основной памяти) является команда **MOV**. За кодом команды следует *источник* данных, в качестве которого может выступать само число, либо элемент памяти, содержащий данные. Затем ставится запятая и указывается *приемник* данных. Мы хотим поместить 0102 в R0:

MOV #102,R0

Первоначальное содержимое R0 будет утрачено. Распечатка всей программы представлена на рис. 2.1.

REGTST RT-11 MACRO VM02-11 26-JAN-79 09:34:07 PAGE 1

```

1          .TITLE  REGTST
2          .LIST   ME
3          .MCALL  .TTYOUT,.EXIT
4
5          000000      R0=%0
6
7 000000 012700 START: MOV      #102,R0
          000102
8 000004          .TTYOUT R0
  000004 110000 .IIF NB <R0>,   MOVB      R0,%0
  000006 104341          EMT      ^O341
  000010 103776          BCS      .-2
9 000012          .EXIT
  000012 104350          EMT      ^O350
10         000000'      .END    START

```

REGTST RT-11 MACRO VM02-11 26-JAN-79 09:34:07 PAGE 1+
SYMBOL TABLE

```

R0      =%000000      START  000000R
. ABS.   000000      000
          000014      001
ERRORS DETECTED: 0
FREE CORE: 18895. WORDS

```

Рис. 2.1. Программа вывода через регистр.

УПРАЖНЕНИЯ. 1. Используйте вместо R0 какой-нибудь другой регистр, чтобы определить, как кодируется регистр, если он выступает в качестве приемника.

2. Можно ли в предыдущей программе использовать любой из восьми регистров?

3. Какова разница между **MOV #102,%1** и **MOV #102,1**? Как транслируется последняя команда? Будет ли она выполняться?

4. Вставьте в предыдущую программу, не внося больше никаких изменений, одну дополнительную команду, чтобы вместо **В** печаталась литера **С**. (Вы уже знаете два разных способа, которые позволяют это сделать.)

5. Нет причин, по которым программа не могла бы изменять собственные команды, хотя это чаще делают ради развлечения, чем для пользы дела. Каков результат выполнения команд

```
START: INC    LABEL
LABEL: MOV    #102,R0
```

Какую команду нужно добавить, чтобы напечатать **В**?

Ввод под управлением монитора. В § 1.4 мы не касались ввода, поскольку не знали, как задать элемент памяти машины в качестве приемника для литеры, набранной на терминале. (Почему этот вопрос не возник для источника данных при выводе?) Однако теперь для хранения вводимых литер у нас появились регистры. Для ввода с терминала одной литеры в системе RT-11 есть макрокоманда **.TTYIN**. За обозначением команды следует элемент памяти, в который монитор должен поместить литеру в коде ASCII. Например,

```
.TTYIN    R0
```

Предыдущее содержимое **R0** будет утрачено. Тот же результат достигается с помощью макрокоманды

```
.TTYIN
```

так как если приемник не указан, то по умолчанию предполагается **R0**. Фактически результатом выполнения **.TTYIN** всегда будет ввод данных в **R0**. Если указан другой элемент памяти, то монитор пересылает вводимую литеру из **R0** в данный элемент памяти. Таким образом, в результате выполнения макрокоманды **.TTYIN R1** оба регистра **R0** и **R1** содержат вводимую литеру, так как команда **MOV** не изменяет содержимое источника данных. Аналогично макрокоманда **.TTYOUT** всегда выводит литеру через **R0**. Таким образом, после выполнения **.TTYOUT R1** регистры **R0** и **R1** содержат выводимую литеру.

Если монитор встретил команду **.TTYIN**, а еще не было набрано никакой литеры, то он переходит в состояние ожидания. Фактически монитор ждет до тех пор, пока не будет набрана вся строка литер, и он не продолжит выполнение программы до того момента, пока не будет нажата клавиша **↵**. Только после этого монитор сформирует *буфер ввода*, который будет содержать все набранные литеры, доступные для использования в программе. Монитор удаляет из буфера ввода предыдущую

набранную литеру при нажатии клавиши RUBOUT, что позволяет исправлять ошибки до ввода строки литер. Всю текущую строку литер он удалит из буфера ввода, если нажать ^U.

УПРАЖНЕНИЯ. 1. Напишите, оттранслируйте и пропустите на машине программу, которая вводит с терминала в R0 литеру и затем выводит ее на терминал (эхо литеры).

2. Измените вашу программу таким образом, чтобы она приглашала ввести данные знаком ?.

3. Измените вашу программу так, чтобы вы могли набирать на терминале любые две литеры (за которыми следует ↵) и получать их эхо. Можно ли в эхо изменить порядок литер? Что произойдет, если вы введете только одну литеру, за которой следует ↵? (Указание: соблюдайте осторожность при использовании регистра R0.)

4. Напишите программу, которая выдает приглашение — знак ? — на ввод одной литеры, за которой следует ↵, посылает эхо литеры, а затем повторяет весь процесс еще раз.

Составим программу прибавления 1 к числу, выбранному нами во время выполнения. Ядром программы будет последовательность

```
.TTYIN
INC      R0
.TTYOUT
```

Помните, что в макрокомандах .TTYIN и .TTYOUT используется R0. Нужно включить эту последовательность в полную программу и убедиться, что при наборе на терминале цифры 5 (за которой следует ↵) программа напечатает цифру 6.

Рассмотрим, что происходит, когда мы вводим цифру 5. Обращение к монитору макрокомандой .TTYIN приводит к изменению содержимого нулевого регистра на 0 65, что соответствует цифре 5 в коде ASCII. Затем после добавления единицы с помощью команды INC R0 в регистре находится 0 66. Обращение к монитору макрокомандой .TTYOUT приводит к выводу на терминал цифры 6.

Следует подчеркнуть, что данная программа работает не с цифрами, которые мы набираем, а с их представлениями в коде ASCII. Она правильно прибавит единицу к цифрам от 0 до 8. Однако при вводе цифры 9 программа выдаст литеру, представление которой в коде ASCII есть 0 72. Проверьте это и самостоятельно установите, какой литере соответствует этот код. Если мы вводим 10, то первая вводимая литера есть 1; поэтому программа выведет цифру 2. Цифра 0 останется в буфере ввода, что вызовет затруднение в работе монитора, когда программа

закончится; во многих системах результатом будет сообщение об ошибке. Даже если мы как-то сможем прибавить единицу последовательно к каждой цифре числа 10, мы все же не достигнем цели — добавления единицы к числу 10 как таковому. Этого и не может произойти, так как мы не указали машине, что 10 есть представление числа в некоторой позиционной системе счисления. А до тех пор 10 понимается как последовательность литер 1 и 0.

Поскольку предыдущая программа оперирует с представлениями в коде ASCII, можно ввести любой символ. Если мы введем А, то выведется В. Результатом ввода В будет С и т. д. Ввод буквы Z (представление в коде ASCII есть 132) вызовет вывод [(представление в коде ASCII — 133).

Ниже приводится фрагмент программы, который ничего не выдает на терминал. Он просто воспринимает набираемые на терминале цифры от 0 до 9 как числа, а не как их представления в коде ASCII:

```
.TTYIN
SUB          #60,R0
```

Вторая команда вычитает 0 60 из содержимого R0. Если мы набираем 7↵, то в результате обращения к монитору с помощью макрокоманды .TTYIN введется соответствующее представление в коде ASCII, равное 67. Команда вычитания преобразует его в 7.

Предположим, мы набрали на терминале 8↵; введется 0 70. Команда вычитания преобразует это в 0 10, что снова есть 0 8.

Со временем вы будете почти автоматически использовать команду SUB для преобразования соответствующего представления цифры в коде ASCII в само число.

Для обратного преобразования числа, находящегося, например, в R0, в код ASCII перед использованием команды .TTYOUT нужно прибавить 0 60. Это делается следующим образом:

```
ADD          #60,R0
```

УПРАЖНЕНИЯ. 1. Напишите программу, которая обеспечивает ввод двух однозначных чисел (за которыми следует ↵) и печатает их сумму, если сумма в свою очередь тоже однозначное число. (Будьте осторожны при использовании R0!)

2. Измените вашу программу так, чтобы можно было вводить $m+n=$, а затем ↵, где m и n — однозначные числа, и на выходе получать их сумму.

3. Напишите программу, которая выдает сумму двух однозначных чисел, являющуюся двузначным числом. (Указание:

какова первая цифра суммы?) Как ваша программа сложит 2 и 2?

Вывод чисел. Команды **ADD** и **SUB** выполняются для любых чисел. Однако, как мы уже видели, возникают сложности при попытке вывести результаты в обычной позиционной системе счисления, в которой мы представляем числа. Предположим, наша программа содержит последовательность

```
.TTYIN      R1
SUB         #60,R1
ADD         R1,R1
```

Обратите внимание, что в третьей строке этой последовательности происходит удваивание содержимого **R1**. Если мы введем 9, то **R1** будет содержать число 18. Однако последовательность **ADD #60,R1** и **.TTYOUT R1** приведет к следующему: **D 18=0 22, 0 22+60=0 102**, а 102 есть код ASCII буквы **B** (проверьте!).

Чтобы вывести число восемнадцать как 18, проанализируем смысл этого обозначения в десятичной системе счисления. Цифра 1 указывает количество десятков в числе 18; цифра 8 — количество единиц. Если *разделить* 18 на 10, то результат будет 1 и 8 в остатке.

В простейших моделях ЦП PDP-11 нет команды деления. Для операции умножения и деления нужна дополнительная микросхема. Однако многие модели PDP-11, хотя и не все, предусматривают наличие этой команды. В оставшейся части данного параграфа предполагается, что операции умножения и деления входят в систему команд; и эту часть вам следует изучить, даже если в вашей модели не предусмотрены такие команды. В § 2.2 мы рассмотрим, как можно умножать и делить, не прибегая к помощи специальных машинных команд.

Вспомним, что в регистре **R1** находится число восемнадцать, которое мы хотим разделить на десять и сохранить остаток. Для этого подходит команда **DIV**:

```
DIV      #12.R0      ; четный регистр!
```

Обратите внимание, что сначала мы пишем число (в восьмеричной системе счисления), на которое хотим разделить. Число, которое мы собираемся делить, *должно быть занесено в регистр*. Кроме того, это должен быть регистр с *нечетным номером*, а в команде **DIV** должен быть указан регистр с *предыдущим номером*; в этот регистр нужно предварительно поместить нуль

(например, с помощью команды «очистки» **CLR**):

```
CLR      R0
DIV      #12,R0
```

Команда **DIV** сохранит частное в регистре с предыдущим номером (в данном примере — **R0**), а остаток — в регистре, который использовался для хранения делимого (**R1**). Итак, в данном случае в **R0** будет 1 и 8 — в **R1**¹⁾.

УПРАЖНЕНИЯ. 1. Напишите программу, которая удваивает вводимые числа до девяти включительно и выдает результат. (Теперь результат удваивания, например, четырех печатается как 08. Почему это происходит?)

2. Измените вашу программу так, чтобы она утраивала вводимые числа.

Для умножения двух чисел одно из них должно быть помещено в регистр с нечетным номером. В этом же регистре будет храниться произведение — результат выполнения команды **MUL**²⁾. Таким образом, содержимое **R1** можно удвоить с помощью команды

```
MUL      #2,R1
```

Теперь можно написать программу вычисления произведения двух однозначных чисел, которые вводятся во время выполнения программы. Последовательность команд для ввода и выполнения вычислений такова:

```
.TTYIN    R1
.TTYIN    R0
SUB       #60,R0
SUB       #60,R1
MUL       R0,R1
```

Заметим, что первое число в команде **MUL** может быть представлено либо содержимым некоторого регистра, либо, как мы только что убедились, в явном виде. То же самое справедливо для делителя в команде **DIV**.

УПРАЖНЕНИЯ. 1. Допишите предыдущую последовательность до полной программы.

¹⁾ Это упрощенное описание команды **DIV**. Полное описание можно найти в приложении Б.

²⁾ Это упрощенное описание команды **MUL**. Полное описание можно найти в приложении Б.

2. Можно ли поменять местами первые две команды последовательности?

Для вывода чисел, состоящих более чем из двух цифр, нужно соответствующее число раз повторить процесс деления на десять и запоминания остатка. Предположим, нам известно, что число может состоять максимум из четырех цифр; в этом случае необходимо разделить число на десять три раза, затем вывести последнее частное и три остатка в нужном порядке. Например, если наше число есть **D 2174**, результат последовательного деления на 10 таков:

	Частное	Остатки
	2174	
После первого деления	217	4
После второго деления	21	7 4
После третьего деления	2	1 7 4

Пусть сначала число находится в **R1**. Тогда команда **DIV** будет использовать **R0** и **R1**. Для остатков нам потребуются еще 3 регистра: будем использовать регистр **R2** для остатка, показывающего число сотен (в нашем примере 1), регистр **R3** — число десятков, **R4** — единиц. Тогда программа вывода начинается следующим образом:

```

CLR      R0
DIV      #12,R0
MOV      R1,R4

```

Напомним, что команда **DIV** поместит остаток в **R1**, где первоначально находилось само число. Подготавливаем следующую операцию деления:

```

MOV      R0,R1
CLR      R0

```

Затем в **R3** помещаем десятки:

```

DIV      #12,R0
MOV      R1,R3

```


Наконец, получаем «сотни» в **R2** и выводим все число:

```

MOV      R0,R1
CLR      R0
DIV      #12,R0
MOV      R1,R2
ADD      #60,R0
.TTYOUT  R0
ADD      #60,R2
.TTYOUT  R2
ADD      #60,R3
.TTYOUT  R3
ADD      #60,R4
.TTYOUT  R4

```

УПРАЖНЕНИЯ. 1. Напишите полную программу вывода числа **D 2174**, которое в начале программы находится в **R1** (придумайте, как его туда поместить).

2. После каждой команды вашей программы выписывайте содержимое первых пяти регистров, которое получается в результате выполнения команды.

3. Обратите внимание, как мы в нашей подпрограмме вывода сэкономили команду **MOV**, сохранив окончательное частное в **R0** вместо пересылки его в **R1**. Можете ли вы еще что-нибудь сократить?

4. Напишите программу, на вход которой поступают четыре однозначных числа, а она выдает их произведение. Будет ли ваша программа работать, если вводимые числа отделить пробелами? Что произойдет, если ввести три числа, за которыми следует **←|**? Что произойдет, если произведение содержит менее четырех цифр?

5. Какое максимальное число можно вывести, используя только регистры **R0÷R5**?

Ввод чисел. Числа, состоящие более чем из одной цифры, вводятся с помощью процесса, обратного процессу вывода. Вместо разбиения числа на отдельные цифры делением на десять сформируем число из составляющих его цифр умножением на **10**. Предположим, нужно ввести **2174←|** и прочитать это при помощи нашей программы как десятичное число. Сначала программа читает цифру **2** посредством последовательности команд

```

.TTYIN   R1
SUB      #60,R1

```

Если больше цифр нет, то этим все закончится. В данном случае это не так, поэтому, когда наша программа встретит **1**, она должна сдвинуть цифру **2** на одну десятичную позицию

влево и добавить 1:

.TTYIN	R0	; берем следующую цифру
SUB	#60,R0	
MUL	#12,R1	
ADD	R0,R1	

На этом этапе **R1** содержит **D 21**. (Каково содержимое **R0**?) Затем следует цифра 7, и мы повторяем эти четыре команды, после чего **R1** содержит **D 217** (почему?). Затем еще одна цифра, и последовательность нужно повторить еще раз: содержимое **R1** в конце концов будет равно **D 2174**. Наш метод заключается в образовании последовательности десятичных чисел

$$\begin{aligned} & \qquad \qquad \qquad 2 \\ & (2 \times 10) + 1 = 21 \\ & (21 \times 10) + 7 = 217 \\ & (217 \times 10) + 4 = 2174 \end{aligned}$$

Заметим, что, хотя числа в последовательных строках таблицы разные, процесс перехода от одной строки к другой один и тот же: умножение на 10 и прибавление следующей цифры. Поэтому, чтобы получить искомое число, можно просто повторить одну и ту же последовательность из четырех команд.

УПРАЖНЕНИЯ. 1. Напишите полную программу, которая вводит четырехзначное число и выводит его.

2. Позволяет ли ваша программа вводить трехзначные числа?

3. Напишите программу, при помощи которой можно вводить два двузначных числа и печатать их произведение.

4. Напишите программу, которая вводит однозначное число, затем трехзначное и печатает:

а) их произведение;

б) частное от деления второго на первое.

Комментарии. В большие программы полезно включать краткие пояснения о назначении команды или группы команд. Особенно это полезно, если программа, написанная одним, должна читаться другим. Комментарию в строке должна предшествовать точка с запятой, которая будет ее первой отличной от пробела литерой. Например, программа может содержать следующее:

; далее следует подпрограмма вывода			
PRINT:	DIV	#12,R0	; O 12 = D 10

Объем комментария — это еще и дело вкуса. Немногие программисты станут включать в свои программы столько комментариев. Вообще говоря, выбор метки **PRINT** устраняет необходимость

дальнейших комментариев относительно назначения подпрограммы. Одни программисты включают комментарии по поводу деления на **O 12**; для других такая часто встречающаяся команда в комментариях не нуждается. Безусловно, комментарий в команде

PRINT: DIV #10,R0 ;восьмеричный вывод

имеет смысл; в противном случае впоследствии при чтении программы самим программистом или кем-либо еще возникает естественное предположение, что была допущена грубая ошибка. Общее назначение строк-комментариев — отмечать переход от одного этапа программы к другому. Комментария заслуживает отдельная команда, если ее функция не вполне ясна и тем более если в ней был применен некий хитроумный прием. Бывает также полезно пояснить использование регистров. Хотя вам предстоит создать собственный стиль работы, все же руководствуйтесь этими общими принципами использования комментариев.

В программах и подпрограммах, приводимых в книге, мы были сознательно весьма скупы на комментарии. Эти подпрограммы служат упражнениями, равно как и иллюстрациями. Сначала каждую из них нужно тщательно изучить, команду за командой, вписывая комментарии там, где все понятно, и оставляя вопросы около тех команд, смысл которых вам неясен. После этого, и не раньше, скопируйте программу в свой файл. Посмотрите, как работает программа с разными входными данными. Если смысл команды все еще туманен, посмотрите, что произойдет, если ее удалить или изменить. Не успокаивайтесь, пока не поймете функцию каждой команды. Наконец, скопируйте программу и сохраните копию с вашими собственными комментариями. Такой подход быстро разовьет ваши способности по написанию программ.

2.2. Команды перехода

Как мы отмечали в § 2.1, да и вы, наверное, уже заметили это сами, в некоторых моделях ЭВМ PDP-11 отсутствуют команды умножения и деления. Предложенная нами процедура вывода десятичных чисел основывалась на последовательном делении на десять; так как совершенно недопустимо потерять возможность выводить числовые данные в десятичном виде, то надо найти какой-нибудь другой способ деления на десять. Даже если в вашей модели есть команды **MUL** и **DIV**, вам все-таки следует изучить предлагаемую здесь процедуру, так как она знакомит с чрезвычайно важными понятиями.

Сущность процесса деления положительного числа на десять заключается в проверке, сколько раз десять «входит» в данное число. Если мы будем вычитать десятки из числа до тех пор, пока это не станет невозможным (получение отрицательного результата исключается), то число, показывающее, сколько раз мы смогли вычесть десять, и есть частное. Например, от семидесяти трех мы можем отнять десять семь раз; поскольку после этого осталось только три, то больше вычитать нельзя. Итак, при делении семидесяти трех на десять семь — это частное, а три — остаток. Все это очевидные вещи, но мы потому заостряем на них внимание, что, когда дело дойдет до программирования этой процедуры, будет необходима полная ясность.

Предположим, что число, которое надо разделить на десять, хранится в **R1**. Займемся вычитанием десяти из содержимого **R1**. Поскольку частное равно числу выполненных вычитаний, мы должны вести их подсчет; подсчет будем вести в **R0**, содержимое которого, следовательно, увеличивается на 1 каждый раз, когда мы вычитаем десять из содержимого **R1**. Таким образом, последовательность команд

INC	R0
SUB	#12,R1

выполняется несколько раз до тех пор, пока следующее вычитание не приведет к отрицательному результату. В итоге мы пришли к очень мощному средству вычислительной машины — способности выбирать различные направления действия в зависимости от результата предыдущего шага. Как правило, это достигается с помощью команды, которая проверяет значение некоторой величины, и в зависимости от этого значения либо перепрыгивает (осуществляет *ветвление*, или *условный переход*) на другую точку в программе, либо нет.

В данном случае мы хотим продолжить вычитание десятков из **R1**, одновременно используя **R0** как счетчик, до тех пор пока в результате вычитания получается положительный (или нулевой) результат. Это делается с помощью последовательности команд

DIVTEN	INC	R0
	SUB	#12,R1
	BPL	DIVTEN

По команде **BPL** (Branch if PLus — ветвление по «плюсу») условный переход происходит в том случае, если результат предыдущей команды — плюс, т. е. положительное число или нуль. В команде **BPL** указывается та команда, которая должна выполняться, если удовлетворяется условие ветвления. Нам нужно повторить последовательность из двух команд, в кото-

рых происходит увеличение **R0**, а из **R1** вычитается десять; поэтому пометим первую строку последовательности и укажем переход на эту строку.

Как только результат вычитания станет отрицательным, не будет удовлетворено условие ветвления, и выполнится та команда, которая в программе идет вслед за **BPL DIVTEN**. Однако мы хотели прекратить вычитание *прежде*, чем результат станет отрицательным; таким образом, последовательность из двух команд выполнилась лишний раз, и это нужно исправить. Используя команду **DEC** (DECrement — уменьшение) для уменьшения содержимого элемента памяти на 1, получим полную подпрограмму деления на десять:

```
· DIVTEN: ·      INC      R0
                  SUB      #12,R1
                  BPL      DIVTEN
                  DEC      R0      ; удаляем лишнее
                  ADD      #12,R1  ; повторение цикла
```

Теперь в **R0** находится частное, а остаток — в **R1**, как и в результате **DIV #12, R0**. Заметьте, что, как и в случае команды **DIV**, перед делением мы должны очистить **R0** (почему?).

Обратите внимание, что **BPL** проверяет значение, полученное в той команде, которая *непосредственно ей предшествует*. Таким образом, порядок следования двух команд в «цикле» **DIVTEN** не может быть изменен.

Это чрезвычайно неэффективный способ деления на десять. Позже мы познакомимся с более удачными методами.

УПРАЖНЕНИЯ. 1. Используя описанный способ деления, напишите программу вывода числа **D 2174**, которое в начале подпрограммы вывода должно быть в **R1**.

2. Сколько команд в действительности выполняется в вашей программе? Проверьте ваш ответ с помощью ЭВМ.

Ввод чисел без использования команды **MUL** достаточно прост. В этом процессе используется умножение на десять, которое можно заменить повторным сложением. Приводим экономный способ умножения на десять содержимого **R1**; в комментариях указано содержимое **R1** и **R2** после выполнения соответствующей команды

```
ADD      R1,R1      ;2X
MOV      R1,R2      ;2X,2X
ADD      R1,R1      ;4X,2X
ADD      R1,R1      ;8X,2X
ADD      R2,R1      ;10X,2X
```


УПРАЖНЕНИЯ. 1. Напишите программу, которая на входе получает трехзначное число, умножает его на семь и выводит результат.

2*. Напишите программу, которая на входе получает два двузначных числа, а на выходе печатает:

- а) большее из них;
- б) их (положительную) разность;
- в) меньшее, точка с запятой, затем большее.

3. Если предварить фрагмент программы **DIVTEN** командой **MOV #12, R2**, то в самом фрагменте можно убрать ссылку на значения делителя. Напишите программу, которая на входе получает трехзначное число, затем однозначное и выводит частное от деления первого на второе.

Ввод чисел. Мы пока еще не можем ввести число, не зная заранее, из скольких цифр оно состоит. Как видно из § 2.1, процесс ввода числа основан на программе, которая берет очередную цифру и десять раз складывает ее с уже вычисленным к данному моменту числом. Вместо того чтобы решать заранее, сколько будет цифр, и писать этот фрагмент соответствующее число раз, мы используем команду условного перехода для повторного выполнения фрагмента в «цикле» до тех пор, пока цифр больше не останется. Весь процесс таков:

```

READ:      .TTYIN
           если не цифра, то переход к DONE
           SUB      #60,R0
           MUL      #12,R1
           ADD      R0,R1
           BR       READ
DONE:      ...

```

Команда безусловного перехода **BR** (BRanch — ветвь) всегда передает управление указанному элементу памяти в программе, не требуя выполнения каких-либо условий. Для краткости мы использовали команду **MUL**, но ее можно заменить и повторным сложением.

Теперь осталось лишь определить, когда будет введена не цифра. Используя несколько команд **BPL**, мы можем проверить, лежит ли код ASCII вводимой литеры в «цифровых» пределах, от 0 60 до 0 71. Это неуклюжий способ: такую проверку проще представить командами, которые нам еще предстоит изучить. А пока мы можем упростить наше задание: договоримся нажимать клавишу специальной *ограничительной литеры* после последней цифры данного числа. Если мы вводим только одно число, то просто нажмем после него, как обычно, клавишу \leftarrow ; два или более чисел будут разделены пробелами. Дело в том, что

как и знак \leftarrow], включающий в себя две литеры

CARRIAGE RETURN— код ASCII 0 15

LINE FEED— код ASCII 0 12

так и пробел, код ASCII которого 0 40, имеют кодовые представления, меньшие 0 60. Поэтому теперь мы можем организовать наш фрагмент программы **READ** так, чтобы он начинался командами

```
READ:      .TTYIN
           SUB      #60,R0
           если меньше, то переход к DONE
```

Допишите этот фрагмент самостоятельно, используя команду **BMI** (Branch if Minus — ветвление по минусу), которая осуществляет переход как раз при тех условиях, когда **BPL** этого не делает; **BMI** является *дополнительной функцией* по отношению к **BPL**. Обратите внимание, что, как и раньше, условие для ветвления основано на результате *непосредственно предшествующей* команды.

УПРАЖНЕНИЯ. 1. Напишите программу, которая выдает приглашение ? ввести число, за которым следует \leftarrow], удваивает число и печатает результат, затем бесконечно повторяет процесс. (Трудность состоит в том, чтобы не воспринимать символ \leftarrow] как часть следующего ввода.) Как вы осуществите выход из этой программы?

2. Напишите программу, которая на входе получает два числа, разделенные пробелом, и печатает

- а) их сумму;
- б) их разность.

В случае б) ваша программа должна уметь работать с отрицательными числами. (Команда **NEG** (NEGate) заменяет содержимое элемента памяти числом с противоположным знаком; используйте команду ветвления для проверки того, является ли результат отрицательным, и, если это так, напечатайте противоположное к нему число, поставив в начале знак —.)

Упражнение 2 дает способ вывода отрицательного числа: нужно вывести соответствующее положительное число, которому предшествует

```
.TTYOUT      #55      ; знак минус
```

Аналогично можно решить вопрос о вводе отрицательного числа: вводим положительное число и отдельно храним информацию о том, что его нужно сделать отрицательным. (В чем конкретно состоит аналогия?) Предположим, мы готовы ввести число, ко-

торое может быть отрицательным. Нужно проверить, является ли первой вводимой литерой —; если это так, то регистрируем этот факт, поместив в **R5** единицу (очистив **R5** перед началом работы подпрограммы). Таким образом, после **.TTYIN** мы должны сравнить содержимое **R0** с числом **0 55**. Для этого используем команду сравнения **CMP** (CoMPare)

```
CMP      R0,#55
```

Команда **CMP** всегда предшествует команде ветвления, действие которой зависит от результата сравнения. **CMP** не изменяет содержимого элемента памяти, указанного в этой команде. В данном случае нам нужна команда **BEQ** (Branch if EQual — ветвление по равенству), чтобы перейти к фрагменту, который оперирует со знаком минус:

```
READ:    .TTYIN
          CMP      R0,#55
          BEQ      MINUS
          отсюда продолжается фрагмент ввода
```

Где-то в программе нам встретится

```
MINUS:    INC      R5
          BR       READ
```

в результате чего мы вернемся к **READ** за следующей литерой.

Того же результата можно достичь несколько иным способом:

```
READ:    .TTYIN
          CMP      R0,#55
          BNE      PLUS
          INC      R5
          BR       READ
PLUS:    отсюда продолжается фрагмент ввода
```

где **BNE** — команда ветвления по неравенству (Branch if Not Equal). Какой из подходов, по-вашему, лучше? Заметим, что во фрагменте, использующем **BNE**, та часть, которая управляет знаком минус, осуществляет переход обратно на **READ** для ввода первой цифры. Вместо этого можно просто использовать еще одну макрокоманду **.TTYIN**. Какие могут быть преимущества и недостатки у этого способа?

После завершения фрагмента ввода положительное число находится, скажем, в **R1**. Содержимое **R5** определяет, нужно ли оставить число положительным (если **R5** содержит нуль), или сделать его отрицательным (если **R5** содержит 1).

Правильное значение в **R1** получается после выполнения следующей группы команд:

	TST	R5	
	BEQ	POS	;если в R1 положительное
	NEG	R1	
POS:	...		

Команда проверки **TST** сравнивает содержимое указанного элемента памяти с нулем для использования в идущей следом за ней команде ветвления. Как и в случае команды **CMR**, указанный элемент памяти не меняется.

УПРАЖНЕНИЯ. 1. Напишите программу для вычисления выражений вида $m \pm n$, где m и n — числа, а знаки $+$ либо $-$ могут быть введены во время вычисления.

2. Измените ваш фрагмент вывода так, чтобы уничтожались стоящие впереди нули.

3. Напишите программу перевода чисел из восьмеричной в десятичную систему счисления. Пусть ваша программа печатает ! и заканчивается, если вводятся цифры 8 или 9.

4. Выясните, что выполняют команды

CMR	R1,R2
BPL	ELSWHR

Сравните результаты этой последовательности с

SUB	R1,R2
RPL	ELSWHR

5. Напишите фрагмент программы, который читает числа, разделенные пробелами или знаками \leftarrow , и *игнорирует* все другие литеры.

Коды условий. В ЦП есть внутренний регистр, так называемое *слово состояния процессора*, в котором хранится информация, необходимая для выполнения текущей команды. Слово состояния процессора обычно обозначается **PS** (Processor Status word); однако эта мнемоника не распознается ассемблером, да и не так уж часто рядовому программисту приходится обращаться непосредственно к **PS**.

Четыре младших бита **PS** (от 0 до 3) являются *разрядами условий*. После того как ЦП выполнит команду, туда помещается информация о результате, который выработала команда. В **PS** разряд 3 — это так называемый *бит N*, который устанавливается ЦП, если какая-либо команда выдает отрицательный результат, и сбрасывается ЦП, если некоторая команда вырабатывает нуль или положительный результат.

ЦП выполняет команду **BPL** с помощью проверки бита **N**. Если бит **N** сброшен, то ЦП передает управление в программе на ту ячейку, которая указана в команде **BPL**. Если бит **N** установлен, то в ответ на **BPL** ЦП ничего не делает (*воспринимает **BPL** как пустую команду*), просто переходит к следующей команде в соответствии с обычным циклом работы. Результат выполнения команды **BMI** вы, безусловно, можете по аналогии вывести самостоятельно. Все команды, которые выполняют арифметические операции, влияют на содержимое бита **N**. В их число входят **ADD**, **SUB**, **MUL** и **DIV** (бит **N** зависит от знака частного); кроме того, **INC**, **DEC** и **NEG**. Команда **CLR** всегда сбрасывает бит **N**.

Команда **MOV** устанавливает бит **N** в соответствии со значением пересылаемых данных. **TST** устанавливает бит **N** по значению содержимого, указанного в команде элемента памяти.

Команда

CMR **X,Y**

где **X** и **Y** — элементы памяти либо выражения типа «**#** число», устанавливает бит **N** в соответствии с результатом вычитания данных, определяемых **Y**, из данных, определяемых **X**. Таким образом, команда

CMR **#60,R1**

установит бит **N**, если содержимое **R1** *больше*, чем **0 60**, и сбросит его в противном случае. Заметим еще раз, что содержимое **R1** не изменяется. С другой стороны, команда

SUB **#60,R1**

вычитет **0 60** из содержимого **R1** и установит бит **N** в соответствии с новым содержимым **R1**; таким образом, бит **N** будет установлен, если первоначальное содержимое **R1** было *меньше*, чем **0 60**, и сброшен в противном случае. Такое несоответствие результатов команд **CMR** и **SUB**, естественно, является богатым источником ошибок.

Разряд 2 в **PS** называется *битом Z*. ЦП устанавливает его, если в команде получается нулевой результат, и сбрасывает, когда в команде получается ненулевой результат. Команды **BEQ** и **BNE** проверяют бит **Z** и реагируют соответствующим образом.

Обратите внимание, что *сами по себе команды ветвления просто проверяют состояние разрядов условий, не изменяя их*. Так что мы можем использовать команду **BPL**, после которой идет **BNE**, чтобы определить, является ли результат операции *строго положительным*.

Поскольку команды ветвления являются основным инструментом структурирования программ, важно знать, как воздействует каждая команда на разряды условий. Прежде чем двигаться дальше, вы должны быть уверены, что представляете себе результат действия каждой известной вам команды на оба перечисленных разряда условий.

УПРАЖНЕНИЯ. 1. Просмотрите несколько своих программ, проставляя против каждой команды состояние битов **N** и **Z** после выполнения команды.

2. Известно, что в данной программе содержимое **R1** может принимать лишь значения 3, 5 и 7 (восьмеричное). Напишите подпрограмму, которая в каждом возможном случае осуществляет переход соответственно на ветви **VALA**, **VALB** и **VALC**, не изменяя содержимого **R1**.

Подсчет элементов данных. Во всех наших предыдущих примерах программ число элементов данных было известно заранее. В качестве примера того, как избежать этого ограничения, напишем программу, читающую совокупность чисел и вычисляющую их среднее арифметическое. Программа для машины без команд **MUL** и **DIV** приводится на рис. 2.2.

Для вычисления среднего арифметического нужно знать, сколько элементов данных было введено. Это делается с помощью использования регистра в качестве счетчика: его содержимое увеличивается на 1 всякий раз, когда считывается число.

Пусть в качестве разделителя между числами выступает любая нецифровая литера, за исключением **←|**, а **←|** сигнализирует о конце ввода. В программе допускается наличие более одной разделительной литеры между числами, а также разделительная литера перед **←|**. Это освобождает от лишних забот во время выполнения. Таким образом, обнаружив разделительную литеру, программа должна перейти на такой фрагмент, который, прежде чем попытаться прочесть следующий элемент данных, пропускает все дальнейшие разделители.

В самом начале в счетчике **R3** должен быть 0. Фрагмент **READ** читает число, используя регистры **R0** и **R1**. После обнаружения разделительной литеры фрагмент **SEP**: увеличивает содержимое **R3** на 1; добавляет содержимое **R1** к текущей сумме, хранящейся в **R2**; сбрасывает содержимое **R1** в нуль в качестве подготовки к чтению следующего элемента данных; пропускает все следующие разделительные литеры, и, если попадется цифра, передает управление в подходящую точку фрагмента **READ** (почему не в начало **READ**?). Если встретится **←|**, программа

```

.TITLE  MEAN
.MCALL .TTYIN, .TTYOUT, .EXIT

R0=%0      ;текущая цифра
R1=%1      ;текущее число
R2=%2      ;текущее значение суммы
R3=%3      ;счетчик чисел
R4=%4      ;среднее арифметическое

START:  CLR      R1
        CLR      R2
        CLR      R3
        .TTYOUT  #77

READ:   .TTYIN
        CMP      R0, #60
        BMI      SEP      ;если (R0)<60
        CMP      #71, R0
        BMI      SEP      ;если (R0)>71
RE1:    SUB      #60, R0
        ADD      R1, R1      ;умножение (R1) на D10
        MOV      R1, R4
        ADD      R1, R1
        ADD      R1, R1
        ADD      R4, R1
        ADD      R0, R1      ;прибавление последней цифры
        BR       READ

SEP:    INC      R3
        ADD      R1, R2
        CLR      R1
S1:     CMP      R0, #15
        BEQ      MEAN
        .TTYIN      ;есть ли еще разделители?
        CMP      R0, #60
        BMI      S1
        CMP      #71, R0
        BMI      S1
        BR       RE1

MEAN:   CLR      R1
        CLR      R4      ;деление (R2)/(R3)
DIVID:  INC      R4      ;здесь будет частное
        SUB      R3, R2
        BPL      DIVID
        DEC      R4      ;убрать лишнее
        ADD      R3, R2      ;повторение цикла
        ADD      R2, R2      ;округление
        SUB      R3, R2
        BMI      PRINT
        INC      R4

PRINT:  напишите свой фрагмент вывода
        и закончите программу

```

Рис. 2.2. Программа вычисления среднего арифметического совокупности чисел,

переходит на ветвь **MEAN** и выполняет вычисления. Процесс иллюстрируется блок-схемой на рис. 2.3.

Среднее арифметическое вычисляется с округлением до ближайшего целого числа; проанализируйте самостоятельно, как это достигается.

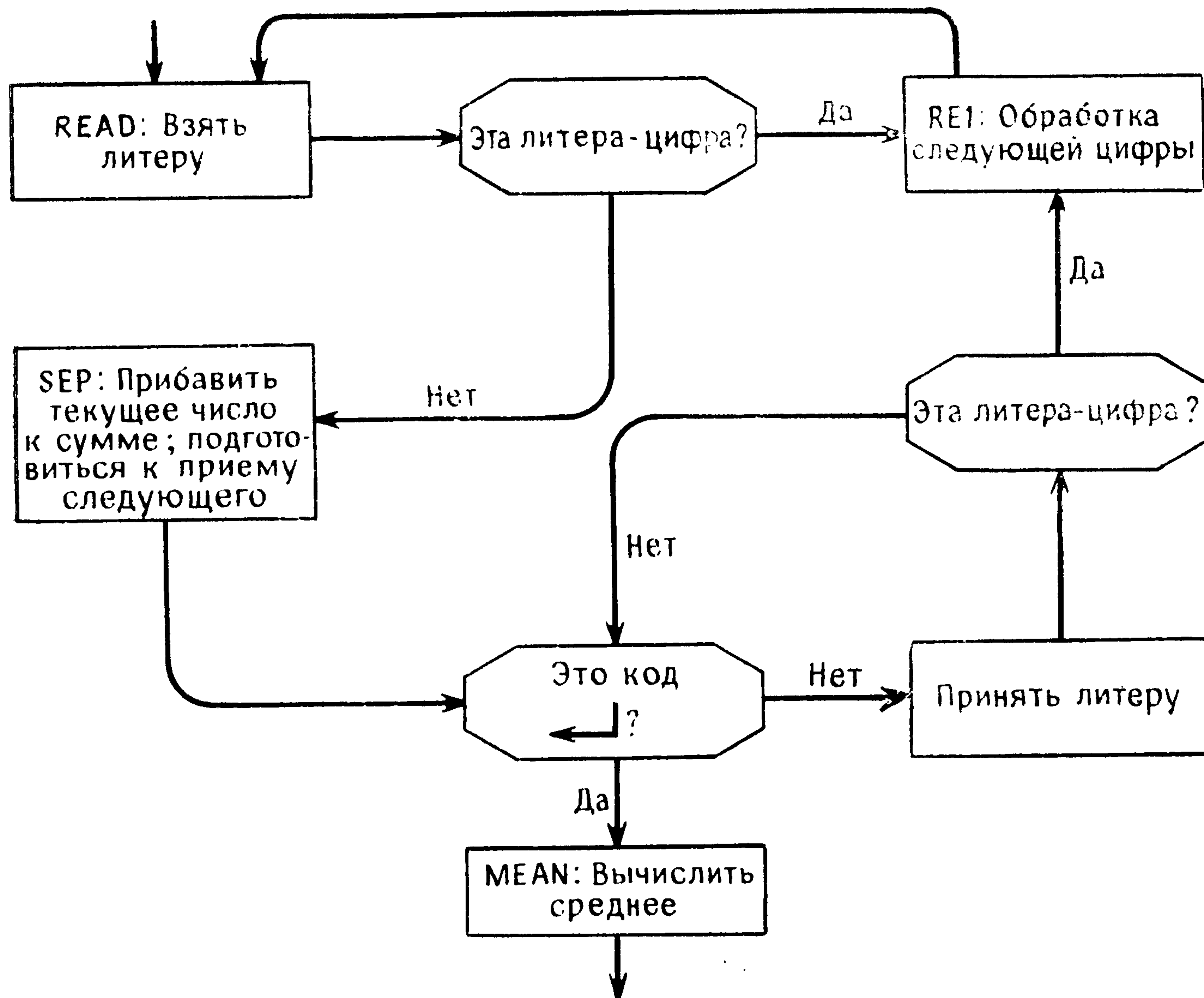


Рис. 2.3. Блок-схема к программе на рис. 2.2.

Команды перехода во фрагменте **SEP** следует изучить с особой тщательностью.

УПРАЖНЕНИЯ. 1. Переделайте ваш фрагмент **PRINT** для программы из рис. 2.2 таким образом, чтобы после печати среднего значения введенных чисел программа снова выдавала приглашение ? для нового ввода.

2. Что произойдет в программе из рис. 2.2, если вы начнете ввод с разделительной литеры? Исправьте ее.

3. Напишите программу, которая читает последовательность чисел и печатает наименьшее и наибольшее из них.

2.3. Память

В § 2.2 мы рассматривали задачу считывания большого количества элементов данных; она была реализована в программе, которая вычисляла среднее арифметическое некоторой совокупности чисел. Мы смогли это сделать, имея в своем распоряжении лишь небольшое количество регистров, просто потому, что нам

не требовалось хранить все элементы данных отдельно; суммирование происходило по мере продвижения вперед. Однако не во всех процессах можно так поступать, и мощность ЭВМ в значительной степени зависит от ее способности хранить и извлекать большие массивы данных. С этой целью используется *память* (или *запоминающее устройство*). Память доступна программисту в виде значительного числа индивидуально адресуемых слов вычислительной машины. В системе разделения времени память требуется одновременно нескольким пользователям, поэтому объем памяти, доступный каждому пользователю, хотя и значителен, но не является неограниченным и не должен расходоваться нерационально.

Место в памяти, необходимое для вашей программы, должно быть в ней специально оговорено. Предположим, мы хотим сохранить содержимое регистра **R1** для дальнейшего использования в своей программе. Нужно придумать собственное имя (по тем же правилам, что и при выборе меток) для того слова памяти, которое мы хотим использовать; назовем его **MEM**. Команда

```
MOV      R1, MEM
```

поместит содержимое **R1** в **MEM**; при этом содержимое **R1** не изменится. Разряды условий устанавливаются в соответствии со значением нового содержимого **MEM**, как если бы ячейка **MEM** была регистром. Фактически в любой рассмотренной нами команде вместо регистров могут использоваться ячейки памяти. Таким образом, мы можем переслать данные из ячейки **MEM** в ячейку **WRD** при помощи команды

```
MOV      MEM, WRD
```

Но этого недостаточно — нельзя просто так использовать в командах программы имена вроде **MEM** или **WRD**. Ассемблер напечатает сообщение об ошибке, так как встретит неописанные символы, и выполнить программу будет невозможно. Мы должны объявить ассемблеру, что нужно зарезервировать слово памяти, и дать ему имя **MEM**. Это можно сделать одной строчкой:

```
MEM:      .WORD      0
```

Директива **.WORD** сообщает ассемблеру о необходимости в коде, сгенерированном для вашей программы, образовать слово, содержимое которого должно быть равно выражению, идущему за **.WORD**. Таким образом, последовательность

```
MEM:      CLR      R0
           .WORD      0
           INC      R1
```

транслируется в три слова: код команды **CLR R0**, слово из всех нулей, затем код **INC R1**.

Таким образом, идея состоит в том, чтобы в программе резервировать слово и пометить его меткой **MEM**. Когда программа загрузится для выполнения, этому слову будет поставлена в соответствие определенная пронумерованная ячейка памяти, содержимое которой станет равным выражению, указанному в директиве **.WORD**. Все это выполняет программа «загрузчик», которая, кроме того, поместит **MEM** в *таблицу распределения памяти* в виде номера соответствующей ячейки памяти.

Вспоминая наши рассуждения относительно ЦП в § 1.3, приходим к выводу, что мы не можем включать директивы **.WORD** в любую понравившуюся нам точку программы. Пусть, например, мы объявили **MEM**, как в предыдущем примере, между двумя командами в программе. После выполнения команды **CLR R0** ЦП попытается *выполнить* слово из всех нулей, образованное директивой **.WORD 0**, как команду. Нулевое слово представляет собой код команды **HALT**; в изолированной системе **HALT** приведет к остановке ЦП, а в системе разделения времени она является командой, недопустимой для обычного пользователя. В любом случае мы снова сталкиваемся с вопросом, обсуждавшимся в § 1.3, обеспечения такого использования памяти, отведенной как для команд, так и для данных, которое не привело бы к путанице.

Операторы резервирования памяти в программе необходимо размещать таким образом, чтобы ЦП ни при каких условиях не принял их за команды. Все операторы такого рода должны предшествовать оператору **.END**, так как он является директивой ассемблеру о прекращении трансляции.

Подходящее место для директив **.WORD** находится между обращением к монитору **.EXIT** и оператором **.END**. При обращении **.EXIT** в операционной системе будет выполняться подпрограмма завершения работы программы; следовательно, нет опасности, что ЦП перейдет к бессмысленному выполнению директив **.WORD** как команд.

Однако не в каждой программе есть обращение **.EXIT**. Так, при отсутствии операционной системы нет и обращения **.EXIT**, и тогда команда **HALT** используется для того, чтобы программа могла остановиться. Кроме того, существуют программы, которые выполняются бесконечно; в этом случае последней командой программы будет команда ветвления. Такая программа никогда не дойдет до обращения **.EXIT**, поэтому нет смысла его туда включать. Тогда мы можем вставить директивы **.WORD** за последней командой, которая непосредственно предшествует оператору **.END**.

Может оказаться, что запросы памяти отдельного фрагмента программы удобно помещать сразу после этого фрагмента. Например, если фрагменту **SEP** (рис. 2.2) требуется область па-

мости, то директивы ее резервирования можно разместить непосредственно перед первой строкой фрагмента **MEM**. (Почему при этом не возникнут только что обсуждавшиеся проблемы?)

Индексация. В § 2.2 мы использовали команды ветвления по условию для решения задачи ввода чисел, состоящих из разного количества цифр. Соответствующая задача вывода немного сложнее. Нужно делить на десять и сохранять последовательные остатки. Когда в результате деления на десять исходное число сокращается до нуля, остатки печатаются, начиная с последнего и кончая первым. Для усвоения этого метода проверьте его на нескольких примерах. Сложность заключается в том, что мы не знаем, как велико может быть выводимое число, и поэтому не можем заранее предсказать количество ячеек, необходимых для последовательности остатков.

Эту проблему мы можем решить с помощью *индексации*. Можно выбрать *любой регистр* — назовем его **Rn** — и использовать его в качестве *индексного регистра*. Регистр **Rn** будет предназначен не для хранения остатка в процедуре вывода, а для адреса ячейки памяти, в которую попадет очередной остаток. Например, пусть в **R1** находится число, которое мы хотим поместить в ячейку **MEM**. Прежде всего мы должны содержимое **Rn** установить равным адресу ячейки **MEM**. Теперь ассемблер будет транслировать каждое обращение к **MEM** как число, равное адресу ячейки, которая зарезервирована под именем **MEM**. Тогда мы можем переслать этот адрес в **Rn** с помощью команды

MOV #MEM,Rn

Обратите внимание на разницу между

(a)	MOV	MEM,Rn
(б)	MOV	#MEM,Rñ

(a) пересылает *содержимое MEM* в **Rn**,

(б) пересылает *адрес MEM* в **Rn**.

После **MOV #MEM,Rn** команда

MOV R1,(Rn)

отправит содержимое **R1** в ячейку **MEM**. Запись **(Rn)** означает, что содержимое **R1** пересылается не в сам регистр **Rn**, а в ячейку, адрес которой представлен содержимым **Rn**. Обратите внимание на разницу между

(a)	MOV	R1,Rn
(б)	MOV	R1,(Rn)

(a) пересылает содержимое **R1** в регистр **Rn**,

(б) пересылает содержимое **R1** в *ячейку памяти, на которую указывает Rn*.

В обоих случаях содержимое **R1** не меняется.

Безусловно, более простым способом пересылки данных будет **MOV R1, MEM**. Однако преимущества индексации проистекают из возможности увеличивать на каждом шаге содержимое **Rn**, выстраивая остатки в последовательность.

Предположим, что число, которое нужно напечатать, находится в регистре **R1**. В процессе деления нам потребуется **R0**; для простоты предположим, что в нашем распоряжении есть команда **DIV**. Используем **R2** в качестве индекс-регистра, а ячейки, начиная с **MEM**, — для остатков.

Для начала установим содержимое **R2** равным адресу **MEM**. Поскольку нам предстоит использовать **R2** в качестве *указателя*, представим себе, что в начале он указывает на **MEM**. Каждый раз после деления на десять и пересылки остатка в память мы увеличиваем содержимое **R2**, чтобы он указывал на следующее слово памяти.

Вспомните, что, как мы отмечали в § 1.3, не только каждое слово памяти ЭВМ PDP-11, но и каждый байт, или полуслово, имеет числовой адрес. Так как адреса слов представляют собой четные числа, то

чтобы индекс-регистр указывал на следующее слово, нужно прибавить 2 к его содержимому.

Процесс пересылки остатков и увеличения содержимого **R2** на 2 повторяется до тех пор, пока не будут найдены все остатки (когда это произойдет?). Теперь нужно выполнить процесс в обратном порядке, уменьшая **R2** на 2 и печатая число, содержащееся в ячейке, на которую указывает **R2**, пока не будет напечатан последний остаток. Заметим, что мы должны внимательно следить за тем, чтобы регистр **R2** всегда указывал в нужное место; в этой ситуации в адресе легко ошибиться на единицу. Фраг-

	MOV	#MEM, R2	
L1:	CLR	R0	; частное в R0
	DIV	#12, R0	; остаток в R1
	ADD	#60, R1	; преобразовать в код ASCII
	MOV	R1, (R2)	; и сохранить
	ADD	#2, R2	; увеличить указатель
	MOV	R0, R1	; для следующего деления
	BNE	L1	; коды, установленные по (R1)
L2:	SUB	#2, R2	; уменьшить указатель
	MOV	(R2), R0	; можно использовать
	.TTYOUT		; .TTYOUT (R2)
	CMP	#MEM, R2	
	BNE	L2	; если еще осталось

продолжение программы
после завершения вывода

мент для вывода приведен на с. 72. Обратите внимание, что в цикле **L1** удалось сэкономить одну команду **СМР** для ветвления. (Можем ли мы это сделать в цикле **L2**?) В комментариях для обозначения содержимого **R1** использовалась запись **(R1)**. Что является основанием для такого сокращения?

В программе, включающей в себя этот фрагмент, нельзя ограничиться только

```
MEM:      .WORD      0
```

так как для хранения остатков может потребоваться несколько слов. Однако директива **.WORD** может использоваться для резервирования нескольких слов памяти, если в ней через запятую указано требуемое содержимое. В нашем случае достаточно пяти слов (почему?), поэтому мы можем написать

```
MEM:      .WORD      0,0,0,0,0
```

Тем самым резервируется блок из пяти слов, первое из которых помечено меткой **MEM**.

Поскольку нас не интересует первоначальное содержимое блока **MEM**, то вместо **.WORD** можно использовать директиву **.BLKW** для резервирования блока слов (**BLock of Words**). В этой директиве указывается число требуемых слов в восьмеричной системе счисления

```
MEM:      .BLKW      5
```

Обычно исходное содержимое этих слов равно нулю, но на это не следует полагаться.

УПРАЖНЕНИЯ. 1. Напишите фрагмент программы для пересылки данных, содержащихся в ячейке, на которую указывает **R0**, в ячейку, на которую указывает та ячейка, на которую в свою очередь указывает **R1**.

2. То же, что и в упр. 1, только **R0** нужно заменить на **MEM**, а **R1** — на **WRD**.

3. Что, по-вашему, делает команда **MOV R1,(R1)**? Так ли это в действительности?

4. Напишите программу, которая считывает последовательность чисел, затем вычисляет, сколько чисел этой последовательности меньше их среднего арифметического, и печатает результат.

5. Напишите программу, которая считывает текстовую строку, состоящую из прописных букв алфавита; затем печатает «закодированную» версию этой строки, заменяя каждую букву в

приведенной ниже таблице буквой, стоящей под ней.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
T	H	E	Q	U	I	C	K	B	R	O	W	N	F	X	J	M	P	S	V	L	A	Z	Y	D	G

Измените вашу программу таким образом, чтобы она воспринимала пробелы и знаки пунктуации и затем печатала их без изменения.

Отладка. В некоторых упражнениях этого параграфа требуется писать довольно сложные программы. В их числе фрагмент ввода, содержащий цикл по пересылке данных из регистров в память; выход из цикла должен быть точно определен, чтобы избежать образования фальшивых данных или потери истинных данных. Во фрагменте вывода решается аналогичная проблема, но в обратном направлении. Кроме того, все вычисления, выполняемые программой, должны получать данные *точно* в том месте, где их оставил фрагмент ввода, и размещать данные *именно* туда, откуда фрагмент вывода может их забрать. Обычно требуется проявить большое искусство при организации сложных программ, чтобы добиться правильной координации их действий. Нужно признать, это — довольно утомительное занятие. Неизбежным следствием всего этого является тот факт, что написание программы без ошибок — редкая и удивительно счастливая случайность. Иногда ошибки видно невооруженным глазом. При работе над этой книгой мы написали программу, в которой некоторый фрагмент очищал блок ячеек памяти, начиная со старшего адреса в блоке к младшему. В **R0** хранился младший адрес в блоке, а **R2** использовался как указатель. Мы собирались выйти из подпрограммы с помощью команд

CMF	R2,R0	
BPL	REPEAT	; взять следующее

Случайно вместо **R0** написали **(R0)**. Содержимое младшего адреса в блоке оказалось меньше адреса, с которого была загружена первая строка программы. Поэтому программа должна была стереть самое себя строка за строкой. Однако очень быстро процесс разрушения сам попал в губительный цикл, и, так как мы работали на изолированной машине, ЦП остановился (почему?).

Ошибки далеко не всегда заявляют о себе столь явно. Даже избегая таких грубых ошибок, как только что описанная, очень легко организовать цикл так, что он будет выполняться лишний раз и очистит на одну ячейку больше. Если это будет последняя ячейка большого блока, зарезервированного для ввода данных, то ошибка обнаруживается только при вводе максимально до-

пустимого количества данных; в программе, выполняющей сложные вычисления, она и в этом случае может оказаться замаскированной. Программы математического обеспечения входят в число наиболее сложных в употреблении, и скептически настроенные профессионалы-программисты утверждают, что вообще не существует программ математического обеспечения без ошибок.

Однако стоическое принятие неизбежности ошибок мало утешает, когда вы сталкиваетесь с программой, которая не работает так, как ей следовало бы. В большинство операционных систем PDP-11 входит системная программа ODT-11 (On-line Debugging Technique — инструмент оперативной отладки), предназначенная для выявления ошибок. Программа ODT весьма полезна и описывается в приложении А, которое вам следует изучать параллельно с § 2.4.

В некоторых системах есть собственная версия DDT — отладочной программы, разработанной для ЭВМ DECsystem-10. Если этот прекрасный помощник есть в вашей системе, скорее научитесь им пользоваться. Полное описание DDT можно найти в книге Michael Singes, Introduction to DECsystem-10 Assembler Language (Wiley, 1978).

Программные средства отладки позволяют выполнять программу постепенно и по мере продвижения вперед проверяют, чтобы она делала то, что должна. Простейшая версия этого процесса *трассировки* существует и независимо от отладочных средств. Пусть, например, вы хотите убедиться, что вводимые данные достигли пункта назначения в памяти. Исправьте вашу программу так, чтобы она выполнялась до определенной точки, но не дальше, вставив в эту точку обращение **.EXIT**. Затем, вооружившись *листингом* своей программы, выполните ее. Обращение **.EXIT** не повлияет на содержимое памяти, которое можно просмотреть, используя специальную команду монитора **E** (Examine — исследовать). Если ваша программа загружена так, что, скажем, **МЕМ** соответствует ячейка **0 1376**, то в операционной системе RT-11 команда монитора

E 1376↵

позволит распечатать (восьмеричное) содержимое **МЕМ**. Таким образом вы можете проверить выполнение любой части своей программы. Можно распечатать содержимое блока ячеек памяти, указав первый и последний адреса блока, отделенные знаком —

E 1370–1376↵

Листинг программы поможет вам определить загрузочный адрес **МЕМ**, указав адрес **МЕМ** относительно начального адреса.

Так, например, ссылка на **МЕМ** в команде может быть обозначена как **000376'**; вспомним, что адрес отсчитывается относительно начального адреса **000 000'**; таким образом, *перемещаемый адрес* есть **376**. Загрузочный адрес получается путем прибавления к нему адреса, по которому будет загружена первая строка программы; это так называемая *константа перемещения*. Как правило, константа перемещения равна **0 1000**, но, во всяком случае, вы теперь прекрасно справитесь с написанием программы, которая печатает собственный начальный адрес.

Иногда полезно протестировать программу, начав ее с определенных ячеек памяти, содержимое которых известно. Для занесения данных в память можно использовать команду монитора **D** (Deposit — занести). Нужно напечатать ячейку и требуемое содержимое, разделенные знаком **=**. Например, установим содержимое ячейки **0 2000** равным **0 10**:

D 2000 = 10 ↵

Можно поместить данные в блок ячеек памяти, набрав на клавиатуре начальный адрес блока и знак **=**, а затем, через запятые, требуемое содержимое слов. Например, в результате выполнения команды

D 2002 = 1,2,3,4 ↵

ячейка **0 2002** будет содержать **1**, ячейка **2004** — **2**, **2006** — **3** и **2010** — **4**. Обратите внимание, что как в **D**, так и в **E** должны указываться адреса слов (т. е. четные числа).

Когда вы убедитесь, что фрагмент программы полностью отлажен, включите в свои комментарии точное описание его действий. Например, в фрагменте ввода можно указать

```
;фрагмент считывания с терминала
;последовательности чисел, любых разделителей,
;возврат каретки завершает ввод. Использует R0 для
;ввода, числа подсчитывает в R1, индексация при
;занесении в память в R2. Требуется обнуления R1,
;R2 указывает на первую ячейку.
```

Фрагмент с такой аннотацией легко модифицировать для использования в других программах.

Автоматическое продвижение указателя. В предыдущем фрагменте вывода мы использовали в цикле последовательность команд

```
MOV      R1,(R2)
ADD      #2,R2
```


для «перешагивания» через ячейки памяти, в которые данные были занесены. Данные настолько часто хранятся в блоках из последовательно идущих ячеек, что с таким шагающим процессом приходится постоянно встречаться при программировании. Аппаратура PDP-11 разработана таким образом, чтобы увеличение указателя происходило при выполнении любой команды, использующей регистр в качестве индекс-регистра. А именно пусть **Rn** используется в команде как индекс-регистр; это значит, что в команде можно обнаружить **Rn** в такой записи: **(Rn)**. Далее, если мы заменим **(Rn)** на **(Rn)+**, то команда выполнится, как и раньше, кроме того, содержимое **Rn** увеличится, и он будет указывать на следующую ячейку. Таким образом, предыдущую шагающую последовательность из двух строк можно заменить одной командой

MOV R1,(R2)+

Обратите внимание на порядок действий: сначала выполняется пересылка данных, затем увеличивается содержимое **R2**. По форме команды ЦП определяет (позже мы исследуем этот вопрос более подробно), что **R2** используется как указатель слов, а не байтов; следовательно, без какого-либо вмешательства программиста происходит увеличение не на 1, а на 2. Такой способ доступа к ячейке, на которую указывает **R2**, называется *автоинкрементной адресацией*.

Достаточно разумно иметь еще и *автодекрементную адресацию* для перешагивания через ячейки памяти в обратном направлении. На этот раз обозначение таково: **—(Rn)**, откуда хорошо виден порядок действий: сначала уменьшение **Rn**, затем выполнение команды. Иначе говоря, при выполнении команды будет использовано новое значение **Rn**. Таким образом, в нашем фрагменте вывода последовательность

SUB #2,R2
MOV (R2),R0

можно заменить одной командой

MOV —(R2),R0

Как и в случае автоинкрементной адресации, ЦП заботится о величине уменьшения.

Отложим до § 2.4 обсуждение команд с автоинкрементной и автодекрементной адресациями, в которых обе ячейки адресуются через один и тот же регистр. До того момента мы предлагаем вам не использовать команды типа **MOV (R2)+, (R2)**.

УПРАЖНЕНИЯ. 1. Напишите программу, которая просматривает блок слов памяти, устанавливая содержимое каждого слова равным его адресу.

2. Напишите программу пересылки содержимого блока из ста слов, начинающегося с **MEM**, в аналогичный блок, начинающийся с **WRD**. Каков адрес последнего слова в каждом блоке?

3. В блоке из ста слов, начинающемся с **MEM**, находятся адреса ста элементов данных. Напишите фрагменты для программы:

а) суммирования всех элементов данных;

б) пересылки каждого элемента данных в то слово блока **MEM**, которое перед этим указывало на данный элемент.

4. Сто различающихся между собой элементов данных хранятся в блоке, который начинается с **MEM**. Те же самые элементы данных, но в другом порядке размещены в аналогичном блоке, начинающемся с **WRD**. Напишите фрагмент программы для замены каждого элемента данных блока **MEM** на адрес, который этот элемент имеет в блоке **WRD**.

Сортировка. Чтобы проиллюстрировать использование блоков ячеек памяти, напомним программу, которая считывает последовательность целых чисел и затем печатает эти числа в порядке возрастания. Этот процесс требует перегруппировки элементов данных в числовом порядке. Такое упражнение является хорошей подготовкой к решению часто встречающейся задачи организации последовательности слов в алфавитном порядке. Существует несколько различных методов для *сортировки* данных; какой из них самый эффективный, зависит от конкретных условий. Метод, который мы будем использовать, достаточно эффективен и довольно прост для программирования.

Для того чтобы упорядочить группу чисел по возрастанию, представим их себе в виде одной длинной цепочки на странице. Начиная слева, двигаемся по цепочке вправо. Каждое текущее число сравниваем с правым соседом. Если сосед больше либо равен, двигаемся на один шаг вправо. В противном случае меняем местами два числа и двигаемся на один шаг влево, чтобы посмотреть, не нужно ли опять переставить меньшее число (если мы уже добрались до самого левого числа, снова двигаемся вправо).

Например, начиная с

2 1 7 5 3

получим такую последовательность шагов:

1	#	2		7		5		3
1		2	*	7		5		3
1		2		5	#	7		3
1		2	*	5		7		3
1		2		5	*	7		3
1		2		5		3	#	7
1		2		3	#	5		7
1		2	*	3		5		7
1		2		3	*	5		7
1		2		3		5	*	7

В каждой строке между двумя числами, которые только что сравнивались, мы поставили #, если они поменялись местами, и *, если нет. Легко видеть, что, как только мы сравниваем последние два числа и обнаруживаем, что их не надо менять местами, процесс завершен.

Все это может показаться слишком сложным, однако ядро программы составляет простой фрагмент **COMPAR**, который сравнивает два числа и, возможно, меняет их местами. Пусть наши числовые данные находятся в блоке ячеек, начиная с **МЕМ**.

```

                                MOV     R0, R2
COMPAR:  CMP     R1, R2
                                BEQ     DONE
                                MOV     (R2)+, R3
                                CMP     (R2), R3
                                BPL     COMPAR
SWAP:    MOV     (R2), R4
                                MOV     R3, (R2)
                                MOV     R4, -(R2)
                                SUB     #2, R2
                                CMP     R2, R0
                                BPL     COMPAR
                                ADD     #4, R2
                                BR      COMPAR

```

Рис. 2.4. Программа сортировки последовательности чисел в порядке возрастания.

Используем регистр **R2** для указания текущего места в блоке, т. е. в качестве индекс-регистра. Таким образом, на каждом этапе мы будем сравнивать содержимое (**R2**) с содержимым слова, следующего за (**R2**); помните, что (**R2**) означает содержимое **R2** и в нашем случае представляет собой адрес данных.

Полностью часть программы, выполняющая сортировку, приводится на рис. 2.4. Для удобства предполагалось, что адрес **МЕМ** находится в **R0**, а адрес последнего элемента последовательности — в **R1**.

Задачи сортировки встречаются так часто, что вам следует потратить какое-то время на то, чтобы до конца уяснить, как работает этот фрагмент. Возможно, вы захотите попрактиковаться в его применении для расстановки в числовом порядке

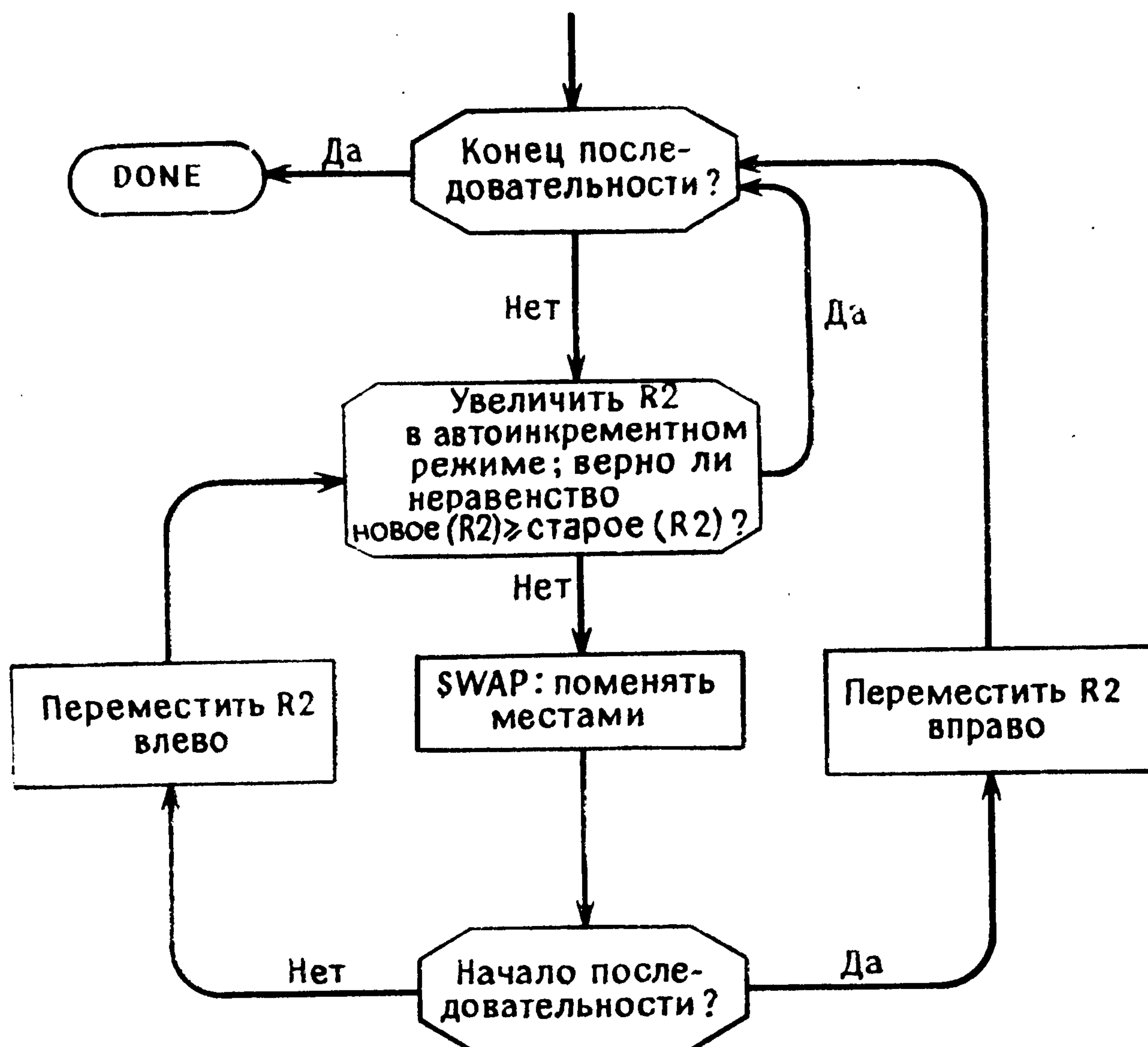


Рис. 2.5. Блок-схема к программе на рис. 2.4.

перетасованных игральные карты или перепутанных букв в алфавитном порядке. Когда будете этим заниматься, внимательно следите за содержимым «регистров». Здесь вам может помочь блок-схема на рис. 2.5.

УПРАЖНЕНИЯ. 1. Напишите полную программу, которая получает на входе последовательность чисел, сортирует их и затем печатает в порядке возрастания.

2. Данные хранятся в ячейках, начиная с **МЕМ**. Число ячеек задается содержимым регистра **R**. Напишите фрагмент программы, который

а) удаляет из памяти все нулевые элементы, перемещая на их место идущие следом данные. Естественно, нельзя изменять порядок следования ненулевых элементов данных, кроме того, нельзя увеличивать их количество, сохраняя на прежнем месте перемещаемый элемент;

б) обнуляет все данные, которые идут после нулевого элемента данных.

3. Напишите программу, которая считывает два числа и печатает их частное с точностью до ста десятичных знаков с округ-

лением. (Указание: научите ЭВМ делению «столбиком» из курса начальной школы.)

4. Напишите программу, которая считывает два числа и печатает период десятичной дроби их частного (например, $3/7 = 0.428571$, в периоде 6 знаков).

2.4. Формат слова

Мы уже знаем, что в команде

INC Rn

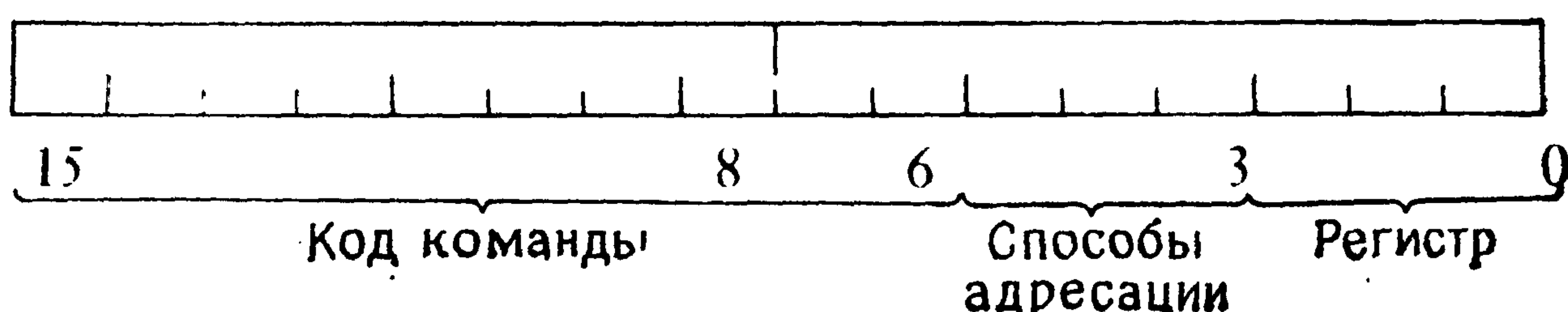
где **Rn** — один из регистров, ссылка на **Rn** кодируется в правом конце слова. Иначе говоря, в листинге для команды **INC R4** будет указан такой код: **005204**. Это восьмеричный код, где самая правая цифра, как вы легко можете убедиться, всегда дает номер регистра. Четыре левые цифры **0052** представляют собой код команды **INC**.

Напомним, что восьмеричная запись не более чем удобный способ представления состояния 16-разрядного слова PDP-11. Самая правая восьмеричная цифра представляет три крайних правых разряда от 0 до 2. Следующие четыре восьмеричные цифры изображают последовательно разряды с 3 по 5, с 6 по 8, с 9 по 11 и с 12 по 14. На представление шестой, и последней, восьмеричной цифры слова PDP-11 остался только один бит, поэтому эта цифра может быть либо 0, либо 1. Таким образом, **0052** (код команды **INC**) состоит на самом деле из 10 битов.

Остается еще одна восьмеричная цифра, об основной функции которой можно догадаться, просматривая листинг команд, адресуемых регистр разнообразными способами, описанными в § 2.3:

005201	INC	R1
005211	INC	(R1)
005221	INC	(R1) +
005241	INC	-(R1)

В общем случае мы имеем следующий формат слова для команд типа **INC**, **DEC**, **NEG**, **CLR** и **TST**, которые обращаются лишь к одному элементу памяти — регистру либо ячейке (*одноадресные команды*)



Для удобства в изображениях формата слова мы указываем границы восьмеричных цифр и точку середины слова.

При кодировании одноадресной команды разряды с 3 по 5, представленные второй восьмеричной цифрой справа, задают *способ (режим) адресации*. Каждому из восьми возможных значений соответствует свой режим адресации, благодаря чему машина PDP-11 располагает необычайно богатыми средствами доступа к данным. Каждый режим имеет название. К настоящему моменту нам известны следующие:

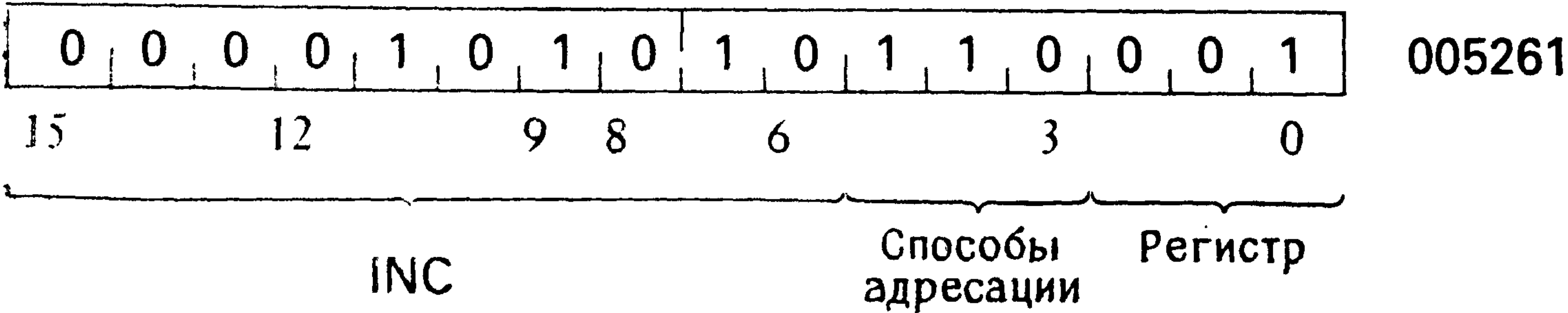
Номер режима адресации	Двоичный код	Название	Мнемоника ассемблера
0	000	Регистровый	Rn
1	001	Регистровый косвенный	(Rn)
2	010	Автоинкрементный	$(Rn) +$
4	100	Автодекрементный	$-(Rn)$

Режим 6 называется *индексным режимом*; команда

INC MEM(R1)

прибавляет 1 к содержимому ячейки памяти с адресом, равным сумме *содержимого R1* и *адреса MEM*. Например, если **MEM** было загружено в ячейку 1376 и **R1** содержит 100 (оба числа восьмеричные), то **MEM(R1)** образует адрес 1476. Вычисление адреса, указанного в команде (*вычисление исполнительного адреса*), выполняется во внутренних регистрах ЦП и не оказывает влияния на содержимое **R1** или **MEM**.

Поскольку **R1** — это регистр и используется способ адресации 6, то ассемблер образует такое слово:



Мы можем рассматривать это как команду **INC ?(R1)**, но так как были использованы все разряды слова, то некуда поместить информацию о том, что ? соответствует **MEM**. Ассемблер поместит ссылку на **MEM** в следующее слово программы. Итак, в PDP-11

одна команда языка ассемблера может занимать более одного слова.

Возможно, этот факт вам уже известен в результате изучения листингов программ. Заметим, что в листинге при широком формате вывода на печать все слова, относящиеся к данной команде, будут расположены на той же строке, что и сама команда. В нашем примере второе слово команды **INC MEM(R1)** транслируется в виде *перемещаемого* адреса **MEM**. Если он был равен, скажем, **376**, то листинг программы может выглядеть так:

```
005261    INC          MEM(R1)
000376
```

Слева мы опустили колонки, в которых изображается номер строки в программе и перемещаемый адрес, по которому находится сама команда.

Проследим, что происходит, когда ЦП встречает эту команду. Распознав код **0052** как одноадресную команду **INC**, он переходит к вычислению исполнительного адреса для определения местонахождения единственного операнда, используя код **61** в качестве основы. Цифра **1** обозначает регистр **R1**; **6** задает индексный режим. Индексный режим заставляет ЦП поместить содержимое **R1** во внутренний регистр. Затем ЦП увеличивает счетчик команд **PC** (Program Counter) так, чтобы он указывал на следующее слово, выбирает содержимое этого слова и прибавляет его к содержимому своего внутреннего регистра. Затем выполняется команда **INC**, причем содержимое внутреннего регистра используется в качестве адреса слова, к которому прибавляется единица. Снова увеличивается **PC**, который теперь указывает на первое слово следующей команды.

УПРАЖНЕНИЕ. Почему ЦП не прибавляет константу перемещения к слову, в котором происходит обращение к **MEM**?

При вычислении исполнительного адреса ЦП сначала проверяет в слове разряды 4 и 5, определяя способ адресации. Эти

Разряды 5 4		Использование содержимого регистра	Мнемоника ассемблера
0	0	как адреса	Rn
0	1	как указателя с последующим прибавлением 1	$(Rn) +$
1	0	после вычитания 1 как указателя	$-(Rn)$
1	1	как индекса для содержимого следующего слова	$MEM(Rn)$

разряды предписывают, как указано выше в таблице, поступать с содержимым регистра, определяемого по разрядам с 0 по 2. Заметим, что никакая комбинация двоичных разрядов не позволяет объединить автоматическое прибавление (вычитание) 1 с индексацией, т. е. формы типа **MEM(R1)+** недопустимы.

С помощью информации, полученной из разрядов 4 и 5, и ссылки на регистр из разрядов с 0 по 2 ЦП вычисляет адрес только что описанным способом. Затем он проверяет разряд 3 — так называемый *разряд косвенной адресации* в одноадресной команде. Если разряд 3 нулевой, то ЦП воспринимает полученный адрес как исполнительный и выполняет операцию с содержимым ячейки, имеющей этот адрес. Такие действия реализуются в режимах 0 (регистровый), 2 (автоинкрементный), 4 (автодекрементный) и 6 (индексный).

Однако если в разряде 3 установлена единица, то вычисленный адрес не является исполнительным. Вместо этого он воспринимается ЦП как *указатель* исполнительного адреса. Другими словами, ЦП заменяет вычисленный к настоящему моменту адрес на *содержимое*, находящееся по этому адресу; это содержимое и рассматривается в качестве исполнительного адреса.

Пусть, например, вычисление исполнительного адреса для команды **INC** перед проверкой состояния разряда 3 дало результат **1376**. Предположим, что текущее содержимое ячейки **1376** равно **2000**. Если разряд 3 нулевой, то исполнительный адрес действительно есть **1376**, и тогда содержимое ячейки **1376** увеличится в данном случае с **2000** до **2001**. Однако если в разряде 3 единица, то **1376** — не исполнительный адрес. В этом случае содержимое ячейки **1376** есть исполнительный адрес, т. е. он равен **2000**. Таким образом, в нашем примере увеличивается содержимое ячейки **2000**, а ячейка **1376** остается без изменения.

УПРАЖНЕНИЕ. Встречались ли нам уже примеры *косвенной* адресации?

Ассемблер распознает символ **@** как признак косвенной адресации. Следовательно, в режимах 1, 3, 5 и 7 используются соответственно такие обозначения: **@Rn**, **@(Rn)+**, **@—(Rn)** и **@MEM(Rn)**. В режиме 1, как мы уже видели, ассемблер допускает запись **(Rn)** вместо **@Rn**.

Ясно, что, чем больше возлагается на ЦП при выполнении команды, тем дольше он будет ее выполнять. Самый быстрый режим — это режим 0, в котором указанный регистр сам является исполнительным адресом. Среди способов прямой адресации индексный режим является более медленным по сравнению с автоинкрементным и автодекрементным режимами; более того, индексный режим требует дополнительных затрат при трансляции и загрузки дополнительного слова. При косвенной адресации

тратится дополнительное время на выборку данных из памяти, однако регистровый косвенный режим по скорости не отличается от автоинкрементного или автодекрементного. Квалифицированный программист должен учитывать время выполнения и длину программы. Например, если в цикле требуется команда **INC MEM(R1)**, то предпочтительнее, по-видимому, иметь перед входом в цикл команду **ADD #MEM, R1**, а в цикле команду **INC (R1)**. Позже мы увидим, что **ADD #MEM, R1** занимает два слова, т. е. мы увеличим длину программы на одно слово. Однако если цикл должен выполняться много раз, то такая замена на более быструю команду оказывается оправданной.

УПРАЖНЕНИЯ. 1. Какова разница между **INC (R1)** и **INC 0(R1)**? Что лучше?

2. В блоке слов, начиная с ячейки **TABLE**, находятся указатели на слова, в которых хранятся адреса ячеек, содержащих данные. Число слов в блоке **TABLE** помещено в слове, на которое указывает **MEM**. Напишите программу, которая стирает все данные.

3. Изучите следующий фрагмент программы:

```

                                MOV      #MEM,R1
                                CLR      R0
LOOP:                          CLR      @0(R1)
                                ADD      #2,(R1)
                                INC      R0
                                CMP      R0,1000
                                BNE      LOOP

```

а) Что в этом фрагменте делается?

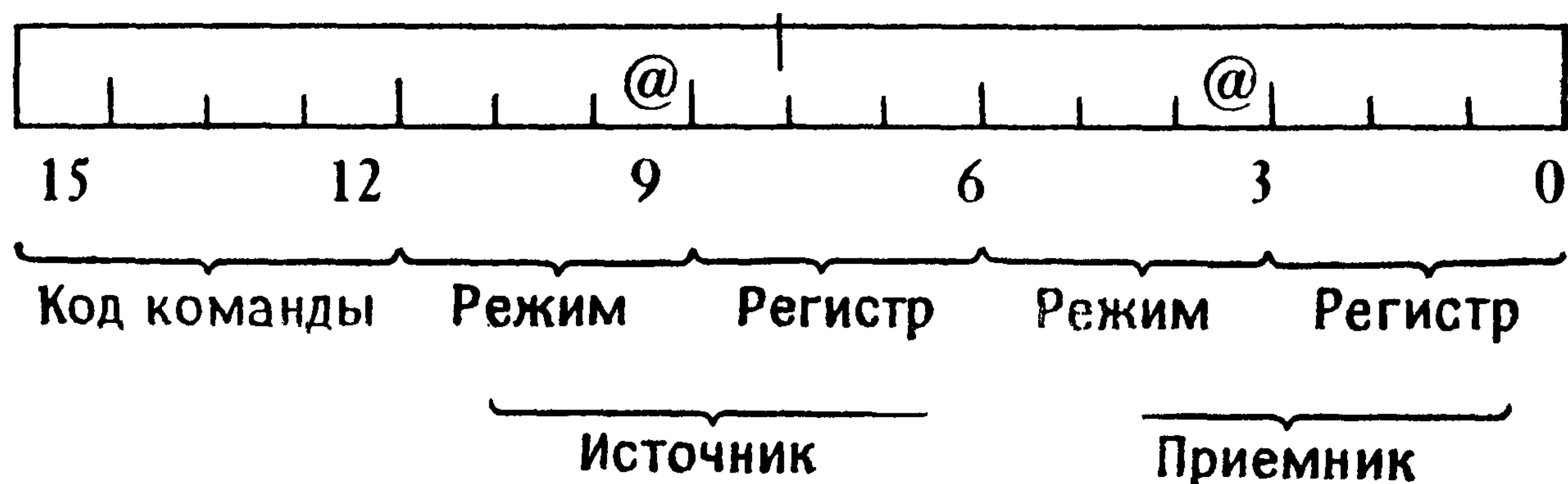
б) Обязательно ли использовать **CLR@0(R1)** вместо **CLR @0(R1)**?

в) Лучше ли станет этот фрагмент, если заменить две строки, начиная с метки **LOOP**, одной командой **CLR @0(R1)+**?

г) Как можно улучшить этот фрагмент?

Двухадресные команды. При изучении одноадресных команд мы пришли к выводу, что для кодирования ссылки на операнд нужно шесть разрядов: три для обозначения регистра и еще три для указания режима адресации. Команды типа **ADD**, **SUB**, **CMR** и **MOV**, в которых адресуются два элемента памяти (*двухадресные* команды), занимают под операнды двенадцать разрядов. Следовательно, в таких командах под код операции отводятся четыре разряда. Первый элемент памяти, указанный в такой команде языка ассемблера, называется *источником*, второй — *приемником*; названия сохраняются даже в команде **CMR**, которая не перемещает данных. Формат слова двухадресной

команды таков:



Любой операнд может быть адресован в режиме 6 (индексный режим) либо 7 (косвенно-относительный режим), как, например,

ADD MEM(R1),@WRD(R2)

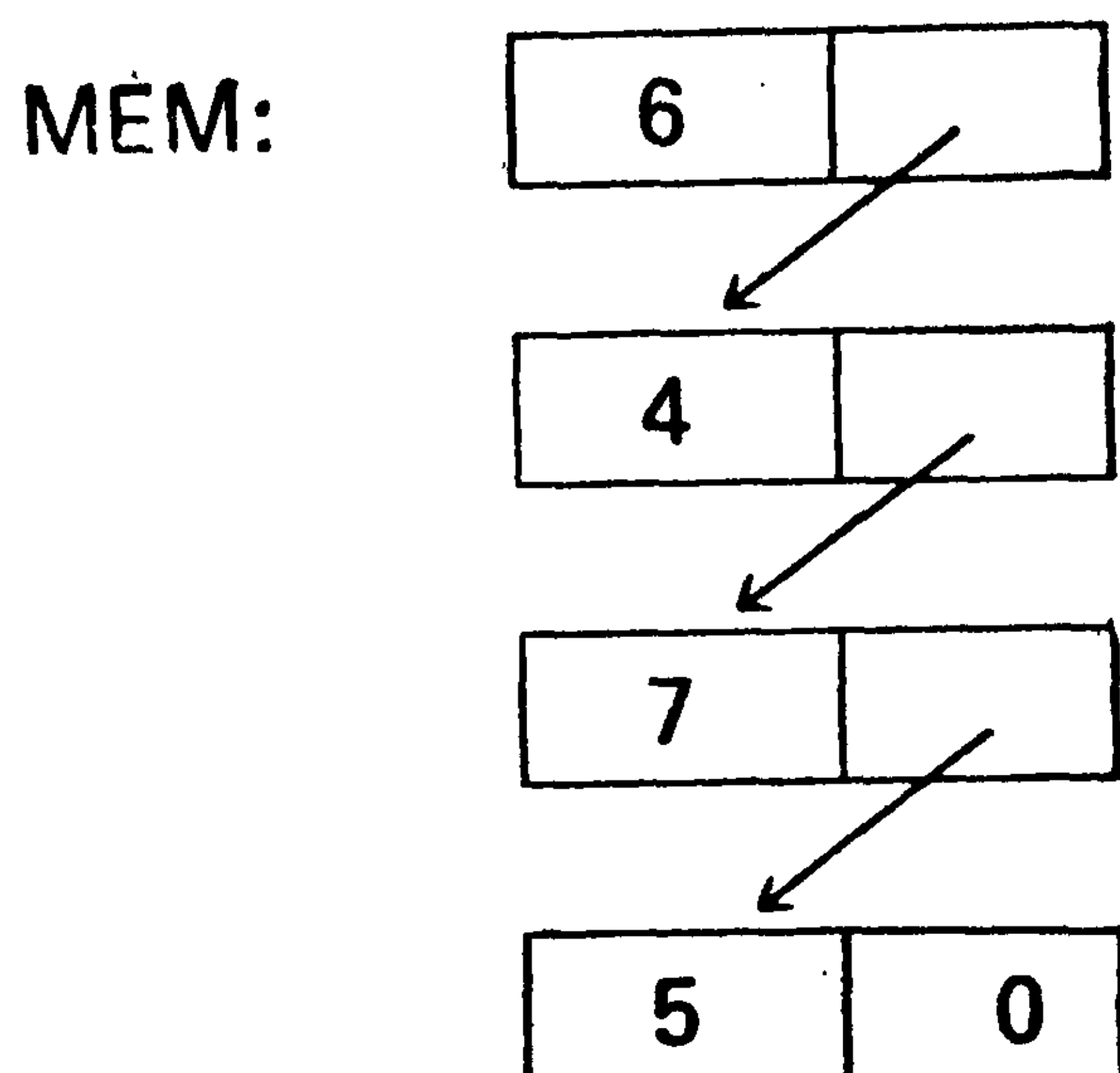
В этом случае первое слово будет 066172, затем идут обращения к MEM и WRD. (Какой код имеет команда ADD?) Обращение к источнику в памяти, если таковой существует, всегда предшествует обращению к приемнику. Таким образом, мы видим, что двухадресная команда может генерировать одно, два или три кодовых слова.

УПРАЖНЕНИЯ. 1. Можете ли вы предложить улучшение команды SUB #2, R1 (эта команда транслируется в два слова)?

2. Данные находятся в памяти в виде *связанного списка*: некоторое количество блоков из двух слов разбросано по памяти; первое слово блока содержит данные, а второе указывает на первое слово следующего блока. MEM указывает на первое слово первого блока; последний блок распознается по нулевому второму слову.

а) Напишите фрагмент программы, который размещает данные в последовательно идущие ячейки, начиная с WRD.

б*) Оставляя на своих местах в связанном списке все элементы данных, разместите в блоке, начиная с WRD, позиции элементов в связанном списке по возрастанию их величины. Например, если связанный список содержит



то блок WRD должен состоять из 2, 4, 1, 3, так как наименьший элемент данных находится на втором месте в списке и т. д.

Для источника и приемника вычисления исполнительного адреса проводятся отдельно. Так как два операнда обозначают разные регистры, то режим адресации одного из них выбирается независимо от другого.

Использование одного и того же регистра в обоих операндах также не приводит к каким-либо затруднениям, если никакой режим адресации не изменяет содержимого регистра. Поблизе познакомившись с порядком выполнения операций в ЦП, мы даже можем допустить, чтобы один или оба режима адресации увеличивали или уменьшали общий регистр. При выполнении каждой команды ЦП проходит через такую последовательность действий:

1. Выбирает из памяти первое слово команды.
2. Увеличивает РС.
3. Вычисляет исполнительный адрес источника (увеличивая РС в режимах 6 и 7).
4. Выполняет автоматическое увеличение или уменьшение содержимого регистра источника.
5. Вычисляет исполнительный адрес приемника (увеличивая РС в режимах 6 и 7).
6. Выполняет автоматическое увеличение или уменьшение содержимого регистра приемника.
7. Выполняет команду, используя адреса, вычисленные на шагах 3 и 5. (*Замечание: однако используется первоначальное содержимое адреса источника.*)

Заметим, что способ адресации, применявшийся для источника, может влиять на вычисление исполнительного адреса приемника, так как увеличение содержимого регистра происходит до того, как вычисляется исполнительный адрес приемника. Рассмотрим, например, команду

MOV (R1)+,(R1)

Предположим, что к моменту выполнения команды R1 содержит 2000, ячейка 2000 содержит 10, а ячейка 2002 пуста. На шаге 3 вычисляется исполнительный адрес источника; он равен 2000. На шаге 4 содержимое R1 становится равным 2002; следовательно, на шаге 5 вычислен исполнительный адрес приемника — 2002. В результате выполнения команды как в ячейке 2000, так и в ячейке 2002 будет 10, а в R1 будет 2002.

Режим адресации источника может повлиять не на вычисление исполнительного адреса, а на *содержимое* по исполнительному адресу приемника. Рассмотрим

ADD (R1)+,R1

с тем же самым исходным содержимым, что и раньше: 2000 в R1 и 10 в ячейке 2000. Исполнительный адрес источника — 2000. После шага 4 R1 содержит 2002; в результате выполнения команды ADD к этому добавится содержимое ячейки 2000, поэтому окончательно R1 будет содержать 2012.

УПРАЖНЕНИЯ. 1. Может ли выбор способа адресации источника в команде оказать воздействие как на вычисление исполнительного адреса приемника, так и на содержимое ячейки с этим вычисленным в конце концов адресом?

2. Попробуйте улучшить программу сортировки из § 2.3.

3. Можно ли получить результат команды ADD #4, R1 более эффективным способом?

Порядок выполнения операций в ЦП таков, что ни при каких условиях режим адресации приемника не может повлиять на вычисление исполнительного адреса источника. Однако порядок выполнения операций сам по себе не сможет защитить *содержимое* по исполнительному адресу источника, когда источник адресуется в регистровом режиме. Рассмотрим с таким же исходным содержимым, как и раньше, команду

ADC R1, (R1) +

На шаге 5 вычисления исполнительного адреса приемника дают 2000; шаг 6 увеличивает содержимое R1 до 2002. Без учета замечания к описанию шага 7 в результате выполнения команды к содержимому ячейки 2000 добавилось бы 2002, т. е. $10 + 2002 = 2012$. Некоторые, хотя и не все, модели ЦП PDP-11 действительно воспринимают такие команды подобным образом. Однако в большинстве моделей ЦП реализовано замечание к шагу 7. В нашем примере, хотя на шаге 6 R1 увеличится и будет содержать 2002, к ячейке 2000 будет добавлено его первоначальное содержимое 2000, в результате чего получится 2010.

Как правило, лучше избегать таких команд, которые не одинаково воспринимаются всеми машинами семейства PDP-11. Если окажется, что подобная команда особенно удобна для использования, обязательно поместите соответствующий комментарий во избежание проблем, которые возникнут, если программа когда-либо будет обрабатываться другим ЦП. При компиляции такой команды ассемблер выдаст сообщение об ошибке, однако это, как правило, не мешает вам ее использовать. В листинге программы напротив команды появится Z в качестве предупреждения, что она не однозначна для всех ЦП. Это один из возможных кодов ошибок, которые обнаруживает ассемблер. Естественно, ассемблер может обнаружить лишь ваше нарушение правил языка ассемблера (*синтаксические ошибки*); он не может

предупредить вас, что программа не выполняет то, что от нее требовалось (*логические ошибки*). Вы, наверняка, уже видели в листингах ваших программ некоторые из следующих кодов:

- A** Неправильный адрес в команде
- E** Нет оператора **.END**; выдается большинством ассемблеров
- M** Одна и та же метка используется несколько раз
- U** Неопределенный символ

УПРАЖНЕНИЯ. 1. Рассмотрите результат выполнения команды **ADD R1,—(R1)** с учетом и без учета замечания к шагу 7.

2. Определите, как ваша система воспринимает подобные команды.

Адресация с использованием счетчика команд. Нам осталось обсудить способы адресации в командах типа **СМР #100, R1** и **CLR МЕМ**, в которых вычисление исполнительного адреса в явном виде не затрагивает регистр. Простота таких операторов языка ассемблера скрывает определенные тонкости в сгенерированном ассемблером коде.

Если в качестве операнда используется само число, как, например, источник в команде **СМР #100, R1**, то такой способ адресации называется *непосредственной адресацией*. Ассемблер транслирует эту команду в два слова, каждое из которых может быть выражено отдельно в виде оператора языка ассемблера.

022701	СМР	(PC)+,R1
000100	.WORD	100

Второе слово просто содержит операнд **0 100**. В первом слове **РС** представляет собой счетчик команд. В PDP-11 счетчик команд — один из общих регистров, доступных пользователю. Фактически это регистр 7, и если ваш ассемблер не распознает обычную для такого регистра мнемонику, что вполне возможно, то в вашу программу должен входить оператор присваивания

РС=%7

Таким образом, непосредственный режим адресации, будучи разбит на свои составные части, представляет собой автоинкрементную адресацию с использованием **РС** и оператор данных. Единственная причина, по которой не используется форма записи кода в две строки, заключается в том, что это меньше соответствует самому назначению кода и занимает больше времени при печати.

Важно четко представлять себе, как работает непосредственный режим адресации. Предположим, что две строки кода ко-

манды **СМР #100, МЕМ** загружены в ячейки 1100 и 1102 (помните, что адреса слов — четные числа). Предыдущая команда установит содержимое **РС** равным 1100. Затем ЦП считывает содержимое ячейки, на которую указывает **РС**, т. е. он считывает команду **СМР (РС)+, R1**. Далее, *перед тем как перейти к выполнению этой команды*, ЦП изменяет **РС** так, чтобы он указывал на следующее слово; в результате **РС** содержит 1102. Необходимо всегда помнить, что **РС** увеличивается (естественно, на 2), прежде чем ЦП приступает к вычислениям, которые предписываются текущим словом команды.

Итак, ЦП определяет в качестве исполнительного адреса источника ячейку 1102; поэтому команда **СМР** сравнит содержимое ячейки 1102 (т. е. 100) с содержимым регистра **R1**. Это как раз то сравнение, которое нам нужно.

Перед тем как выполнить сравнение (но, как мы уже видели, после завершения вычисления исполнительного адреса источника), ЦП *в соответствии с автоинкрементным режимом адресации* увеличит содержимое **РС** на 2. Теперь в **РС** находится число 1104 — адрес первого слова следующей команды. ЦП считывает это слово, увеличивает **РС**, и выполнение программы продолжается.

Тот же результат мог быть получен, хотя и менее эффективно, следующим образом: сначала обнулить **R0**, затем выполнить команду

026001
000100

СМР 100(R0),R1

На первый взгляд может показаться странным, что эта команда увеличивает **РС** сразу на 4 без всякого автоинкрементного режима в команде. Однако соответствующее увеличение **РС** встроено в работу индексного режима. ЦП, как и положено, при считывании первого слова команды **СМР** увеличивает **РС**, чтобы **РС** указывал на второе слово команды, содержащее данные для индексации. Затем в соответствии с индексным режимом ЦП считывает это слово, *увеличивает РС* и выполняет команду **СМР**.

С другой стороны, **СМР #100, R1** — это такая команда **СМР**, которая транслируется в *одно слово*, а исполнительный адрес источника хитроумным способом указывает на следующее слово. Это получается потому, что ассемблер в соответствии с синтаксисом команды помещает индексируемые данные как раз в следующее слово. Команды, состоящие из одного слова, увеличивают **РС** только на 2, поэтому для перемещения **РС** за слово, которое содержит данные, нужен автоинкрементный режим. Это еще один пример тех дополнительных усилий, которые мы должны прикладывать, чтобы ЦП не попытался выполнить слово с данными как команду.

В режиме непосредственной адресации, как видно из команды типа **MOV #MEM, R1**, могут быть адресованы перемещаемые данные. В этом случае трансляция проходит так же, как и раньше, перемещаемой величине отводится слово вслед за командой, и она попадет на свое место во время загрузки.

УПРАЖНЕНИЯ. 1. Каков результат команды **CLR #MEM**?

2. Получится ли **101** после выполнения команды **INC #100**?

3. Каков результат выполнения последовательностей

(a) INC	(PC)	(b) INC	(PC) +
DEC	(PC)	DEC	— (PC)

Относительный режим адресации. Обычно память PDP-11 адресуется в языке ассемблера прямым указанием имен, которые программист приписывает определенным ячейкам памяти, как, например, в командах **CLR MEM** или **MOV MEM, WRD**. Такая форма обращения в память называется *относительным режимом адресации* и тоже использует **PC** в качестве своего регистра. Команда **CLR MEM** воспринимается ассемблером как **CLR X(PC)**, где **X** — число, определяемое ассемблером. Изучение следующего примера покажет нам, каким образом ассемблер должен вычислять **X**, чтобы заставить команду работать. Пусть первое слово команды **CLR MEM** транслируется в ячейку с перемещаемым адресом **100**, а **MEM** имеет перемещаемый адрес **376**. При вычислении исполнительного адреса с использованием **PC** команда **CLR** в индексном режиме увеличит содержимое **PC** дважды, как описано выше; т. е. это вычисление выполняется с **PC**, содержащим перемещаемый адрес **104**. Величина **X** должна быть такой, чтобы после добавления к **104** получалось **376**; поэтому **X** будет равно **272**, и команда **CLR MEM** транслируется так, как будто это **CLR 272 (PC)**.

000100	005067		CLR	MEM
000102	000272			
...				
...				
000376	000000	MEM:	.WORD	0

В общем случае **X** должно быть равно

(адрес **MEM**) — (адрес слова, следующего за ячейкой, содержащей **X**),

так как, когда ЦП вычисляет исполнительный адрес, в **PC** находится адрес слова, следующего за тем, в котором помещается **X**. Число **X** может быть во втором или третьем слове команды в зависимости от того, какой операнд адресуется в относительном режиме; ассемблер автоматически сделает необходимые корректировки.

Обсуждение проводилось в терминах перемещаемых адресов, хотя ЦП, естественно, оперирует с адресами, которые получились после загрузки. По существу, в этом нет никакой разницы хотя бы потому, что если ассемблеру не дать никаких указаний относительно места загрузки программы, то он может сформировать лишь перемещаемые адреса. Нет разницы еще и потому, что, как мы видели, ассемблер определяет, что именно надо поместить во второе слово команды типа **CLR MEM**, вычитая один адрес программы из другого и образуя *относительный* адрес двух слов; отсюда название этого способа адресации. Расстояние между двумя адресами не изменится, если оба сместятся на одну и ту же величину после прибавления общей константы перемещения. Заметим, что само по себе это расстояние является *абсолютным*, а не перемещаемым.

УПРАЖНЕНИЕ. Оттранслируйте вручную следующую программу, предполагая, что при загрузке метка **START** будет иметь адрес 1000, а также зная, что команда ветвления занимает одно слово:

START:	INC	MEM
	INC	MEM
	ADD	#1, MEM
	BR	START
MEM:	.WORD	0
	.END	START

Код, сгенерированный для команд в относительном режиме адресации, является примером *позиционно-независимого кода* PIC (Position-Independent Code). Это название может показаться странным, так как мы видели в последнем примере, что код зависит от положения команды в программе. Однако он не зависит от того, с какого места программа будет загружена в память; как было замечено, разность между двумя адресами в программе не меняется при перемещении всей программы. В системах разделения времени есть смысл писать системные программы целиком в позиционно-независимом коде. Эти программы одновременно принадлежат многим пользователям; неразумно было бы жонглировать областями памяти каждого пользователя, с тем чтобы давать таким программам всякий раз один и тот же загрузочный адрес, либо размещать позиционно-зависимый код по разным загрузочным адресам. Аналогичные рассуждения применимы в любой системе к программе, которая загружается после пользовательских программ переменной длины; к числу таких программ принадлежат программы отладки.

Отрицательные числа. В заключение этого параграфа кратко обсудим представление отрицательных чисел в языке ассемблера PDP-11 и их кодирование.

В операторах языка ассемблера отрицательные числа представляются обычным образом с помощью знака —. Например,

	MOV	#-1, -2(R1)
или		
MEM:	.WORD	--1

Заметьте, что $-2(R1)$ означает число -2 , модифицированное содержимым $R1$ (*отнюдь не* содержимым со знаком минус). Это удобный способ перешагивания назад на одно слово от слова, на которое указывает $R1$, не меняя содержимого $R1$.

Отрицательные числа, безусловно, могут получаться и в результате вычислений. Представление отрицательного числа в слове ЭВМ PDP-11 не зависит от того, каким образом оно там оказалось. Разряд 15 слова называется *знаковым разрядом* и устанавливается равным 1 для отрицательных чисел. Однако, как вы легко можете убедиться (каким образом?), представление, скажем, 1 сильно отличается от представления -1 , а не только нулем или единицей в разряде 15. В архитектуре PDP-11 для представления отрицательных чисел используется так называемый *дополнительный код двоичного числа* (или *двоичное дополнение*). Если X — положительное число, то для образования представления $-X$ нужно:

1. Образовать представление X ; так как X — положительное число, то знаковый разряд нулевой, а разряды с 0 по 14 содержат двоичный код X .

2. Вычесть 1.

3. Заменить все нули единицами, а единицы — нулями.

С учетом такого соглашения наибольшее положительное число, которое может храниться в слове PDP-11, имеет 0 в разряде 15 и 1 во всех остальных. Его восьмеричное представление есть 077777, а десятичное значение $2^{15}-1=32\,767$.

Заметьте, что -1 представляется единицами во всех разрядах: 177777, -2 есть 177776 и т. д. Мы имеем следующий спектр значений:

Положительные	Наибольшее	077777
		077776
		...
		...
Отрицательные	Наименьшее	000001
	Наибольшее	000000
		177777
		177776
		...
		...
		100001
	Наименьшее	100000

Обратите внимание, что при правильном использовании мы описываем -2 как число, меньшее -1 , и т. д.

Аналогия с движущимся в обратном направлении автомобильным спидометром может немного прояснить смысл этой таблицы значений. Во всяком случае, бóльшая осведомленность вскоре сделает ее менее загадочной.

УПРАЖНЕНИЯ. 1. Как представить отрицательное число **0 100000** в виде слова PDP-11?

2. Опишите систематизированный метод получения восьмеричного кода представления отрицательного числа, исходя из положительного числа **X**.

3. Каков результат выполнения команд

CMP	#-77777, #77777
BM!	MINUS

В чем, по-вашему, здесь загвоздка?

4. Если в **R0** находится **7777**, то каков результат выполнения команды **INC R0**?

5. Метка **MEM** соответствует ячейке с перемещаемым адресом **100**. Как будет кодироваться команда **CLR MEM**, если при трансляции она будет располагаться, начиная с ячейки с перемещаемым адресом **376**?

3. СТРУКТУРА ПРОГРАММЫ

3.1. Подпрограммы

В гл. 2 уже обсуждалась блочная структура наших программ. Основополагающая идея состоит в разбиении решаемой задачи на небольшие подзадачи. Затем для каждой подзадачи пишется фрагмент программы, и вся программа становится объединением таких фрагментов, которые различным образом связаны между собой. В программе они выполняются в определенном порядке. Обычно порядок бывает таким: ввод, вычисления, вывод.

Однако более сложные задачи могут потребовать создания *ветвящихся* программ, т. е. таких программ, в которых из разных фрагментов можно попасть в любой другой фрагмент. Например, нам может понадобиться на различных этапах исполнения печатать результаты. И нам вряд ли удастся обойтись всего одним фрагментом для печати результатов, поскольку в момент окончания вывода оказывается утерянным адрес ячейки, где встретила команда перехода на этот фрагмент. Значит, вычисления не могут быть продолжены с той стадии, на которой они были оставлены. Причина затруднения заключается в том, что фрагмент для печати результатов, как только он написан, становится неотъемлемой частью программы. Переход к нему возможен с любого места, а выход один — он раз и навсегда зафиксирован в программе.

Конечно, мы могли бы в конце фрагмента поставить команды условного перехода и с их помощью вернуться к месту, где встретился переход. Однако это было бы чересчур сложно и громоздко. К тому же это было бы и бессмысленно, поскольку в нашем распоряжении имеется возможность создания *подпрограмм*, которые предназначены именно для того, чтобы не возникало подобной трудности.

Нас, разумеется, интересует вопрос, как писать подпрограммы, но прежде мы рассмотрим, что они дают. Допустим, что наш фрагмент для печати результатов был оформлен в виде подпрограммы, первая строка которой имеет метку **PRINT**. Печать осуществляется передачей управления на метку **PRINT**, причем предварительно принимаются меры, чтобы после завершения печати вычисления в программе продолжились с того места,

где они были приостановлены. Эти действия называются *вызовом* подпрограммы.

Благодаря использованию подпрограмм работа по составлению сложных программ значительно упрощается. В первоначальном наброске программы нет необходимости во всех подробностях обдумывать, скажем, ту ее часть, которая печатает результат. Если в какой-то момент нам потребуется напечатать в десятичном виде содержимое ячейки **MEM**, то мы можем, не вдаваясь в подробности, просто написать

PRINT MEM

как будто бы в языке ассемблера существует специальная команда **PRINT** печати числа. К сожалению, такой команды нет, и поэтому перед запуском программы придется заменить эту строку на *последовательность вызова* и написать соответствующую подпрограмму.

Такого подхода необходимо придерживаться всегда, когда какое-то действие должно выполняться много раз. Сначала мы предполагаем, что существует специальная команда, выполняющая это действие. А потом, когда структура программы проработана, наступает время для дополнения ее необходимыми подпрограммами.

Программа редактирования текста. Рассмотрим, как пишутся подпрограммы, на примере создания программы редактирования текста, хранящегося в памяти. Предположим, что нам требуется оставить только по одному пробелу между словами, так что задача нашей подпрограммы — исключить все лишние пробелы. Достигнуть этого можно путем замены кода ASCII для пробела нулем и последующим исключением всех нулевых слов в массиве, содержащем текст.

Осуществить такой план довольно легко и без написания подпрограммы. Для этого подойдет следующая группа команд, в которой предполагается, что **R2** указывает на очередную литеру текста:

	CMP	#40,(R2)+	; встретился пробел?
	BNE	ONWARD	; нет — на продолжение программы
SPACE.	CMP	#40,(R2)	; да — исключить лишний пробел
	BNE	ONWARD	
	CLR	(R2)+	
	BR	SPACE	
ONWARD:	...		

Этот фрагмент непосредственно в таком виде можно включить в главную программу, просматривающую текст. К сожалению, в нем не учитываются все обстоятельства, возникающие при обнаружении пробела во время просмотра. Так, после точки мы можем пожелать оставлять два пробела, удаляя все остальные.

А после символа \leftarrow нам может понадобиться один или несколько пробелов, отмечающих начало абзаца, который начинается с фиксированного отступа, скажем в четыре пробела. Конечно, все эти пожелания можно удовлетворить, добавляя подходящие проверки, предшествующие нашему фрагменту удаления пробелов. Но в результате получилась бы длинная неразрывная последовательность кодов, настолько загроможденная командами переходов, что при всем желании не удалось бы усмотреть хоть какую-то структуру в программе. Чаще всего такой стиль программирования говорит о том, что у программиста не сложилось еще четкого представления о структуре программы. Запутанность в программировании всегда приводит к ошибкам, затрудняет их обнаружение, вызывает раздражение при чтении текста программы и препятствует дальнейшему ее усовершенствованию.

Слишком расточительно также помещать фрагмент, удаляющий пробелы, в большую программу, которая ищет очередной пробел в тексте. Нам может понадобиться исключать пробел и в других случаях, если, к примеру, он предшествует символу \leftarrow или знаку препинания. Безусловно, не имеет никакого смысла повторять ту же группу команд в каждом таком месте.

Все эти рассуждения указывают на необходимость выделять фрагмент, удаляющий пробелы, чтобы к нему можно было обратиться, когда бы это ни потребовалось, но чтобы такие обращения не вносили путаницы при модификациях хорошо структурированной программы. Поэтому мы включим в нашу программу самостоятельную группу команд, например такую:

;подпрограмма исключает лишние пробелы, начиная с (R2)

SPACE:	CMP	#40,(R2)
	BNE	RETURN
	CLR	(R2)+
	BR	SPACE

RETURN: возврат в главную программу

Подпрограмма должна быть написана так, чтобы выход из нее осуществлялся на команду, следующую за командой вызова. Это принципиальный момент в понятии подпрограммы. Любые последующие действия должны быть предприняты в главной программе уже после возврата из подпрограммы. Сам подход, при котором задача разбивается на независимо программируемые компоненты, а логическая структура представляется в виде диаграммы передач управления между ними, настоятельно требует удерживать компоненты от непрошеного вмешательства в дела друг друга.

Итак, возврат должен осуществляться по адресу, содержащемуся в счетчике команд РС в момент передачи управления на

метку **SPACE** (как вы помните, **PC** всегда указывает на *следующую* команду). Поэтому при вызове подпрограммы **SPACE** (а на самом деле и любой другой подпрограммы) требуется сохранять значение **PC**.

Самым подходящим местом хранения является другой регистр. Допустим, что **PC** был сохранен в **R5**, так что возврат должен происходить по адресу (**R5**). Нельзя, однако, воспользоваться командой **BR (R5)**. Такая запись бессмысленна, ибо адрес в команде безусловного перехода должен быть вычислен *ассемблером*, который во время трансляции определяет величину относительного смещения. Перед загрузкой в команде перехода должен стоять фактический адрес; вычисление исполнительного адреса не производится.

Можно, однако, передать управление по адресу, на который указывает **R5**, очень простым способом. Нам требуется только, чтобы счетчик команд **PC** указывал на ту же самую ячейку. Достигается это так:

```
MOV      R5,PC
```

Если возврат осуществить такой командой, то наша программа окажется завершенной.

Теперь изучим процедуру вызова подпрограммы. Нам нужно что-то вроде

```
MOV      PC,R5
BR       SPACE
```

хотя полностью этим наши запросы не удовлетворяются (почему?). Есть, однако, команда, которая сохраняет **PC** в обозначенном в ней регистре и одновременно передает управление по указанному адресу. Это команда вызова подпрограммы **JSR** (Jump to SubRoutine), которой мы теперь и воспользуемся:

```
JSR      R5,SPACE
```

Команда **JSR** кодируется в одном машинном слове, в которое (если читать слева направо) заносится семиразрядный код команды **004**, три разряда отводятся под номер регистра, в котором будет храниться значение **PC**, а шесть разрядов — под смещение. Исполнительный адрес вычисляется, исходя из величины последнего. (Экономится ли *память* при использовании команды **JSR** вместо засылки значения **PC** в регистр и затем применения команды безусловного перехода?) Заметьте, однако, что для хранения **PC** необходимо указать именно *регистр*. Команда сохраняет текущее значение **PC**, которое поэтому является адресом следующей за **JSR** команды, что и требуется.

Если в программе, кроме удаления лишних пробелов, больше ничего делать не нужно, то ее основная часть должна выглядеть так:

LOOP:	CMP	#40,(R2) +
	BNE	LOOP
	JSR	R5,SPACE
	BR	LOOP

При возврате из подпрограммы управление будет передано в точности на команду **BR**. Может показаться заманчивым оформить возврат так, чтобы он сразу происходил на метку **LOOP**, и таким способом сэкономить одну команду. Но по причинам, о которых говорилось выше, это было бы ложной экономией. У подпрограммы свое собственное задание, и она не должна отвлекаться на какую-либо дополнительную помощь главной программе. Более того, в дальнейшем вместо возврата на метку **LOOP** нам может понадобиться выполнить какие-то другие действия по редактированию текста. В таком случае желательно, не затрагивая подпрограмму **SPACE**, просто внести добавления в главную программу. Кстати, после возврата из подпрограммы тогда не потребуется переход на метку **LOOP** для сравнения текущей литеры с пробелом (почему?).

Даже в таком простом примере мы должны тщательно проверить, что данные для обработки в подпрограмме заносятся *в точности* туда, где последняя ожидает их найти. Подпрограмма обнулит все пробелы, начиная с ячейки (**R2**), и возвратит управление, как только ей попадется иная литера. В то же время, когда в главной программе обнаруживается очередной пробел, он может быть лишь первым пробелом после какой-то иной литеры (почему?) и потому должен остаться. Следовательно, после обнаружения пробела значение **R2** в главной программе автоматически увеличивается перед вызовом подпрограммы **SPACE**, так что первый пробел подпрограмма удалить не может.

УПРАЖНЕНИЕ. Воспользуйтесь приведенными выше главной программой и подпрограммой и составьте свою программу чтения строки текста с терминала в память и распечатки ее после удаления лишних пробелов. Прокомментируйте ее и нарисуйте блок-схему.

Вложенные подпрограммы. Усовершенствуем нашу программу редактирования текста так, чтобы она реагировала на точки. Главную программу можно изменить прежде, чем мы решим, как поступать с точками. После выяснения, является ли текущая литера пробелом или нет, вместо перехода на метку **LOOP**

нужно сделать еще один шаг и проверить, не является ли она точкой. Поэтому наша состоявшая из четырех строк программа станет такой:

```

      LOOP:      CMP      #40,(R2)
                BNE      L1
                TST      (R2)+
                JSR      R5,SPACE
      L1:        CMP      #56,(R2)

```

Здесь мы отказались от автоинкрементного режима в команде с меткой **LOOP**, потому что иначе нам пришлось бы в качестве компенсации применить автодекрементную адресацию в команде с меткой **L1**, что внесло бы только путаницу. За это приходится расплачиваться лишней командой, но важнее ясно видеть, на что в текущий момент указывает **R2**. Во всяком случае есть и компенсация — исключается дополнительная проверка после возврата из подпрограммы **SPACE** (какая проверка?).

Теперь можно закончить главную программу:

```

                BNE      L2
                JSR      R4,PERIOD
      L2:        TST      (R2)+
                BR       LOOP

```

Относительно подпрограммы **PERIOD** здесь имеются в виду некоторые соглашения. При обращении к ней считается, что в этот момент **R2** указывает на точку, а при возврате **R2** должен указывать на литеру, предшествующую той, которая будет сравниваться в команде с меткой **LOOP**. Возврат из подпрограммы должен осуществляться через **R4**. Конечно, эти допущения не абсолютны. Если мы найдем целесообразным оформить вход в подпрограмму **PERIOD** или выход из нее по-другому, то всегда сможем внести соответствующие изменения в главную программу. Наиболее удачное согласование обязанностей программы и подпрограммы не всегда достигается с первого раза.

В подпрограмме **PERIOD** будет лишь исключаться пробел перед точкой и разрешаться ровно два пробела после нее. Для простоты будем считать, что после точки всегда следуют по крайней мере два пробела. Подпрограмма исключает лишь превышающие это число пробелы.

Чтобы исключить возможный пробел перед точкой, в подпрограмме **PERIOD** достаточно уменьшить значение **R2** и вызвать подпрограмму **SPACE**; после выхода из последней **R2** вновь будет указывать на точку. Далее нужно сдвинуть **R2** для пропуска двух разрешенных пробелов и вызвать подпрограмму **SPACE**. Перед выходом содержимое **R2** должно быть вновь

уменьшено, чтобы оно оказалось в соответствии с принятым в главной программе соглашением.

PERIOD:	CMP	#40, -(R2)
	BNE	P1
	JSR	R5,SPACE
	TST	-(R2)
P1:	ADD	#10,R2
	JSR	R5,SPACE
	TST	-(R2)
	MOV	R4,PC

Вы можете возразить, что вместо вызова **SPACE** для исключения одного пробела достаточно применить команду **CLR (R2)**. Однако в дальнейшем мы, возможно, пожелаем усовершенствовать программу и не будем оставлять пустые слова, если в них содержится лишний пробел, а будем сдвигать всю оставшуюся

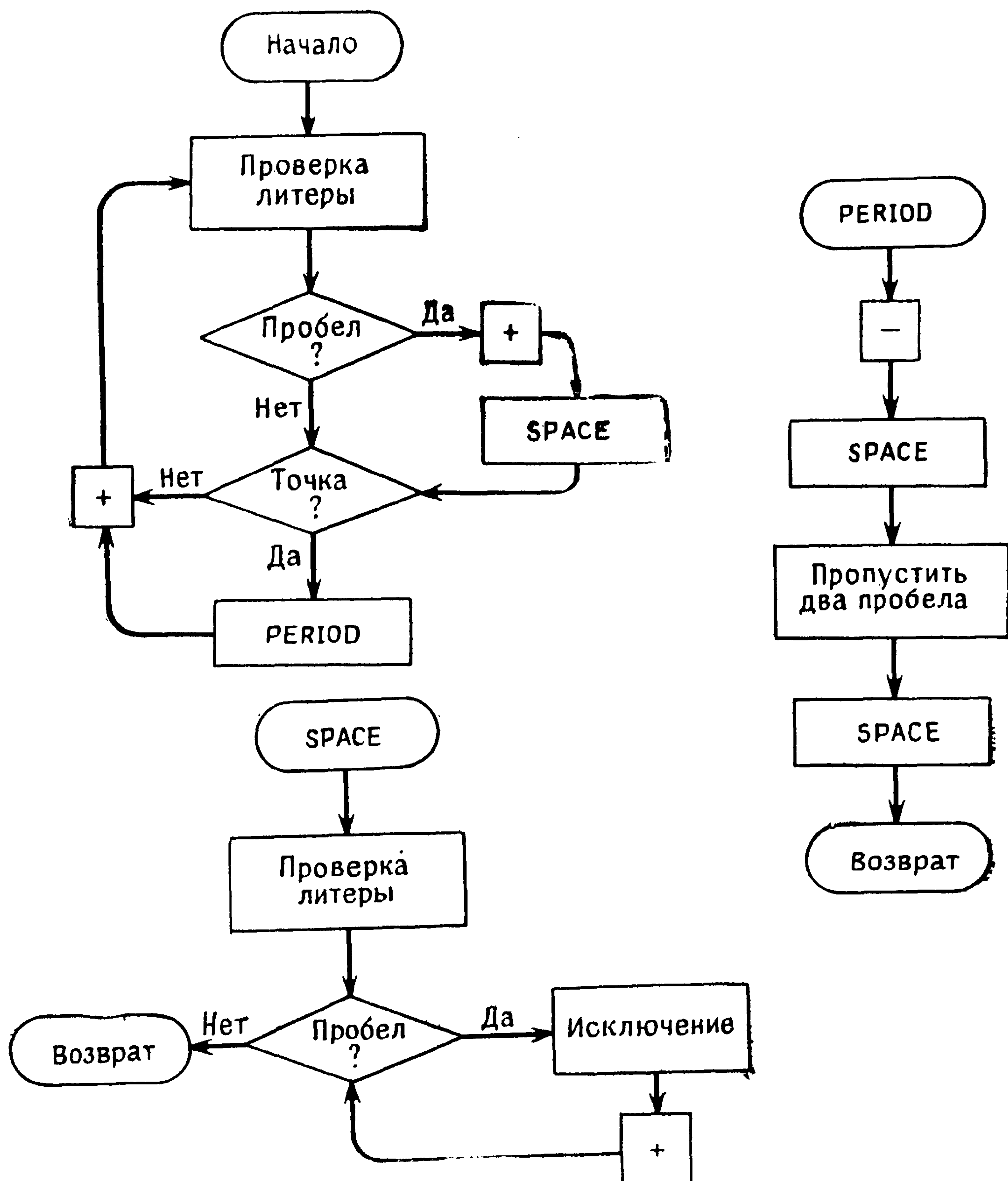


Рис. 3.1. Блок-схема программы редактирования текста,

часть массива данных на одну позицию. Мы заранее предусматриваем такое изменение и потому используем подпрограмму **SPACE** при каждом исключении пробела, так что в дальнейшем будет необходимо изменить только эту подпрограмму.

Блок-схема всей программы показана на рис. 3.1. Символы **+** и **—** отмечают, что **R2** указывает соответственно на одно слово вперед или назад.

УПРАЖНЕНИЯ. 1. Какое серьезное упущение вы заметили в блок-схеме и в самой программе?

2. Допустим, что перед точкой было поставлено несколько пробелов. Подпрограмма **SPACE** исключит все, кроме одного. Будет ли исключен оставшийся пробел в результате выполнения трех первых команд подпрограммы **PERIOD**?

3. Напишите законченную программу. А теперь в корне переработайте ее, чтобы главная программа и подпрограмма как можно больше сочетались друг с другом.

В приведенном примере показано, как одна подпрограмма вызывает другую. Такой способ называется *вложением* подпрограмм. Подпрограмма может быть вызвана из в свою очередь вызванной подпрограммы и т. д. Без сомнения, шесть подпрограмм, каждая из которых вызывает последующую, — большая редкость. В этом случае *глубина* (или *уровень*) вложенности равна шести.

Могло бы показаться, что глубина вложенности ограничена числом доступных нам регистров. Мы использовали регистр **R4** для вызова подпрограммы **PRINT**, помня, что, когда **PRINT** вызывает **SPACE**, пятый регистр необходим для сохранения адреса возврата. Поэтому в подпрограмме **PRINT** должен быть использован другой регистр. Для создания программ, решающих сложные задачи, чрезвычайно важно, чтобы подпрограммы могли вкладываться на любую глубину, так что выделение особого регистра для каждого уровня вложенности было бы чересчур суровым ограничением на возможности системы PDP-11. Неограниченный уровень вложенности между тем вполне реален, поскольку команда **JSR** выполняет не только то, о чем было рассказано выше.

Регистр связи. Аппаратная часть системы PDP-11 обеспечивает эффективный и удобный способ передачи управления между подпрограммами. Если последовательности вызова подпрограммы и возврата из нее рассматривать вместе с аппаратными возможностями, на которые они опираются, то весь процесс называется *связью* подпрограмм. Регистр, который фигурирует в команде **JSR**, называется *регистром связи* или *указателем связи*. Мы уже видели, что ограничение на глубину вложенности сни-

мается, если только на различных уровнях может быть применен один и тот же регистр связи. Именно эта возможность существует в системе PDP-11, потому что в дополнение к другим своим функциям команда **JSR** сохраняет первоначальное содержимое регистра связи. Таким образом, одна команда

JSR R5,SUB

- 1) сохраняет содержимое **R5**;
- 2) заносит значение **PC** в **R5**;
- 3) заносит адрес метки **SUB** в **PC**.

Обсуждение вопроса о том, где команда **JSR** хранит содержимое регистра связи, отложим до § 3.2.

Чтобы увидеть, к чему все это приводит, допустим, что в нашей программе редактирования текста обе подпрограммы (и **SPACE**, и **PERIOD**) используют в качестве регистра связи пятый регистр. При вызове подпрограммы **PERIOD** из главной программы

JSR R5,PERIOD

в пятый регистр заносится текущее значение счетчика команд **PC**, а исходное содержимое **R5** сохраняется в другом месте. Потом в подпрограмме **PERIOD** командой

JSR R5,SPACE

вызывается подпрограмма **SPACE**. Это приводит к засылке в регистр **R5** адреса возврата из подпрограммы **SPACE** в **PERIOD**. В то же время сохраняется предыдущее содержимое **R5**, т. е. адрес возврата из подпрограммы **PERIOD** в главную программу.

Одно дело — знать, что необходимая для выхода из обеих подпрограмм информация сохранена, другое дело — располагать ею. Мы все еще можем вернуться из **SPACE** в **PERIOD** командой **MOV R5,PC**; но поскольку мы не знаем, где в результате исполнения команды **JSR R5,SPACE** было сохранено значение счетчика команд для выхода из подпрограммы **PERIOD** в главную программу, то и не имеем возможности вернуться в последнюю.

Беда в том, что при входе в подпрограмму **SPACE** мы сохраняем некоторую информацию, но не восстанавливаем ее при выходе из подпрограммы. Вместо команды **MOV R5,PC** мы должны применить специальную команду возврата из подпрограммы **RTS** (ReTurn from Subroutine). Команда

RTS R5

- 1) заносит содержимое **R5** в **PC**;
- 2) засылает в **R5** последний, сохраненный командой **JSR** адрес (и еще не восстановленный предыдущей командой **RTS**). Таким образом, команда **RTS** не только возвращает управление главной программе или подпрограмме, лежащей на предыдущем уровне вложенности, но также приводит в исходное состояние все регистры, измененные в результате вызова командой **JSR** текущей подпрограммы. Пара команд **JSR** — **RTS** предоставляет в распоряжение программиста очень удобный и, в сущности, автоматический способ связи между подпрограммами, оставляя ему возможность сконцентрироваться на существе решаемой задачи.

Регистр связи можно использовать в главной программе и для других целей, будучи уверенным, что он будет восстановлен после возврата из подпрограммы. Но для большей ясности имеет смысл некий регистр использовать только как регистр связи. Обычно для этих целей резервируется **R5**. Самый важный вывод из способа применения команд **JSR** и **RTS** таков:

Один-единственный регистр можно использовать для вызова и возврата при работе с любым числом подпрограмм.

Допустим, к примеру, что подпрограмма **SUB1** вызывает **SUB2**, которая в свою очередь вызывает **SUB3**. В результате получим ситуацию, представленную на рис. 3.2. Три команды **JSR** последовательно сохраняют следующие величины:

Адрес возврата из **SUB2** в **SUB1**
 Адрес возврата из **SUB1** в главную программу
 Первоначальное содержимое **R5**

причем *нижняя из них заносится первой*, средняя — второй, а верхняя — последней. Адрес возврата из **SUB3** в **SUB2** нигде не пересылается, поскольку в подпрограмме **SUB3** нет команды **JSR**, и потому следующего уровня вложенности не существует.

Неплохо представить себе, что последовательно вызываемые команды **JSR** аккуратно складывают сохраняемую ими информацию на вершину стека ¹⁾. Имея в виду именно такой образ, мы говорим, что первая в списке сохраняемых единиц информации кладется на дно стека.

¹⁾ От английского *stack* — кipa. — Прим. перев.

Теперь функцию команды **RTS** по восстановлению содержимого регистра можно рассматривать как снятие с вершины стека верхнего элемента и засылку его в регистр **R5**. Следовательно, вновь обращаясь к рис. 3.2 и образу стека, первая команда

<u>Главная программа</u>	<u>SUB1</u>	<u>SUB2</u>	<u>SUB3</u>
...	SUB1: ...	SUB2: ...	SUB3: ...
...
...	JSR R5,SUB2
JSR R5,SUB1
...	...	JSR R5,SUB3	RTS R5
...	RTS R5	...	
...		RTS R5	
...			

Рис. 3.2. Связь подпрограмм.

RTS R5 должна выполняться в подпрограмме **SUB3**. Содержимое **R5**, устанавливаемое командой **JSR** в подпрограмме **SUB3**, указывает на адрес возврата в **SUB2**. Вдобавок эта команда **RTS** снимает с вершины стека и заносит в **R5** адрес возврата из **SUB2** в **SUB1**.

Подпрограмма **SUB2** теперь благополучно продолжает свою работу, завершая ее командой **RTS R5**. Содержимое **R5** (восстановленное командой **RTS** в подпрограмме **SUB3**) указывает на адрес возврата в **SUB1**. К тому же эта последняя команда **RTS** забирает следующий элемент (который является адресом возврата из **SUB1** в главную программу) с вершины стека и заносит его в **R5**.

Наконец, когда подпрограмма **SUB1** заканчивает работу, то командой **RTS R5** управление передается главной программе и восстанавливается первоначальное значение **R5**, которое только и оставалось в стеке к этому моменту.

По-видимому, в таком способе связи подпрограмм самое утешительное то, что его достаточно хоть раз уяснить и больше уже не уделять ему внимания. Если пятый регистр был однажды выбран в качестве регистра связи, то из *любой* подпрограммы возврат должен осуществляться командой **RTS R5**. И, что важно, это вне зависимости от того, насколько сложна последовательность передач управления между подпрограммами.

УПРАЖНЕНИЯ. 1. Напишите подпрограммы для ввода и вывода десятичных чисел. Имеются ли в ваших подпрограммах обращения к другим подпрограммам для выполнения операций умножения и деления?

2. Напишите подпрограмму для печати результатов вычислений главной программы в виде таблиц. Под каждый столбец отведите восемь позиций, а числа выравнивайте по правому краю,

т. е. младшие цифры всех чисел в столбце должны лежать на вертикальной линии. Ваша подпрограмма должна запоминать остатки и затем вызывать подпрограмму для подсчета требуемого количества пробелов и столбцов.

3. Напишите подпрограмму печати таблицы простых чисел, меньших десяти тысяч.

4*. Напишите программу чтения набранной на терминале команды ассемблера PDP-11 и распечатки ее кода, имея в виду определенный фиксированный адрес загрузки. Постарайтесь учесть как можно больше как самих команд, так и применяемых в них способов адресации.

Текст ASCII. Не только в редакторах текста, но и во многих других программах требуется печатать текст в коде ASCII. Очень часто текст представляет собой сообщение, в котором запрашивается ввод информации или, наоборот, выдается информация о ходе выполнения программы. Теперь мы знаем, как написать подпрограмму для печати подобного рода сообщений, но остается еще невыясненным вопрос о том, как изначально загрузить текст сообщения в память, если он не будет вводиться с терминала. Мы могли бы применить директиву **.WORD**, но такое решение оказалось бы слишком громоздким. Для того чтобы загрузить сообщение типа **WAITING FOR GODOT**¹⁾ в блок с меткой **MEM**, нам пришлось бы написать

```
MEM:      .WORD      127,101,111,...
```

и т. д. еще четырнадцать чисел. Если на одной строке для них всех не хватает места, то требуется еще одна (или несколько) директива **.WORD**, поскольку инструкция ассемблера не должна занимать более одной строки.

К счастью, существуют директивы ассемблера, облегчающие нашу задачу. По меньшей мере мы можем не утруждать себя запоминанием или просматриванием таблицы кодов ASCII, потому что знак ' ассемблер интерпретирует как указание на то, что следующий символ нужно воспринимать в коде ASCII. Поэтому предыдущую инструкцию можно заменить на

```
MEM:      .WORD      'W,'A,'I',...
```

и т. д.

Это не лучшее решение, но, прежде чем пойти дальше, заметим, что генерация кода ASCII при помощи знака ' очень удобна в других случаях. Например, зная, что '0 есть то же са-

¹⁾ «В ожидании Годо» — название пьесы С. Бекета. — Прим. перев.

мое, что и 60, вместо

```
CMP      #60,(R1)
```

можно написать

```
CMP      #'0,(R1)
```

не заботясь о правильности кодировки и представляя команду в более наглядном виде. Такая форма записи должна использоваться лишь по отношению к литерам, имеющим графический образ (но никогда по отношению к управляющим символам).

Простейший способ указать ассемблеру, чтобы он занес в память текст в коде ASCII, заключается в применении одной из специальных директив **.ASCII** или **.ASCIZ**. Директива выглядит так:

```
MEM:      .ASCII      /WAITING FOR GODOT/
```

Аналогичный синтаксис и у директивы **.ASCIZ**. Обратите внимание на *ограничители* **/.../**, которые указывают на начало и конец текста, но сами не рассматриваются как его часть. В качестве ограничителей можно использовать большинство обычных литер. Чаще других выбирают литеры **/**, **"** и **'**. Транслятор запоминает, какая литера была ограничителем в начале текста, и заносит литеры в коде ASCII в память машины до тех пор, пока не встретит второй такой же ограничитель.

Учтите, что директивы **.ASCII** и **.ASCIZ** не являются ни командами языка ассемблера, ни вызовами монитора. Они, подобно директиве **.WORD**, представляют собой указания программе-ассемблеру на необходимость кодирования описанным выше способом.

Ни одной из них нельзя ввести текст, превышающий по длине одну строку программы, написанной на языке ассемблера. Но поскольку в директивах **.ASCII** и **.ASCIZ** отдельную литеру можно представить, заключая ее код ASCII в угловые скобки (*вне* ограничителей), текст

```
WAITING  
FOR GODOT
```

можно занести в память, начиная с ячейки **MEM**, следующей директивой:

```
MEM:      .ASCII      /WAITING/(15)(12)/FOR GODOT/
```

Обе директивы заносят текст в память *побайтно*: каждый семибитный код ASCII заносится в семь младших разрядов те-

кущего байта. Представление в памяти предыдущего сообщения будет таким:

A	W	MEM
T	I	MEM + 2
N	I	MEM + 4

и т. д.

Нетрудно написать подпрограмму печати сообщения, хранимого в памяти в последовательности байтов. Если начать с команды **MOV #MEM, R1**, то после вызова

.TTYOUT (R1)

на терминале появится буква **W**. Код буквы **A**, хранящийся в старшем байте ячейки **MEM**, не приведет к путанице, потому что вызов монитора **.TTYOUT** пересылает из ячейки в регистр только байт, а затем переходит на подпрограмму внутри монитора, которая печатает содержащуюся в регистре литеру. Системная макрокоманда **.TTYOUT**, осуществляющая вызов, содержит команду **MOVB** пересылки байта (MOVE a Byte):

MOVB (R1), R0

Источником в этой команде является младший байт ячейки, адрес которой хранится в **R1** (т. е. ячейки **MEM**). (Не забывайте, что адрес слова совпадает с адресом его младшего байта.) Приемником же служит младший байт регистра **R0**. Байтовая команда, в которой используется регистровый способ адресации, всегда обращается к младшему байту указанного регистра. Старший байт регистра не имеет адреса, и поэтому ассемблер интерпретирует запись **R0+1** как ссылку на регистр **R1**.

Чтобы напечатать следующую букву, **A**, регистр **R1** должен указывать на следующий байт, т. е. на старший байт ячейки **MEM**. Следовательно, мы можем вывести на терминал цепочку побайтно хранимых литер при помощи

```

LOOP:      .TTYOUT    (R1)
           INC        R1
           BR         LOOP

```

Чтобы **R1** стал указывать на следующий байт, его содержимое, как видно, каждый раз нужно увеличивать на единицу. Более простой путь, однако, такой:

```

LOOP:      .TTYOUT    (R1)+
           BR         LOOP

```

Ссылка на регистр **R1** при расширении макрокоманды встречается в команде

```
MOV B      (R1)+,R0
```

ЦП при выполнении этой команды сначала перешлет данные, а потом *увеличит R1 только на 1, поскольку MOV B — команда байтовая*. Байтовые команды более подробно будут рассмотрены в § 3.3.

Другой способ занесения данных в память побайтно состоит в использовании директивы **.BYTE**, синтаксис которой аналогичен синтаксису директивы **.WORD**:

```
MEM:      .BYTE      'W','A','I',...
```

и т. д. Еще один способ заключается в применении символа " в директиве **.WORD**, благодаря чему последующие две литеры будут преобразованы в код ASCII и занесены в два байта одного слова:

```
MEM:      .WORD      "WA","IT","IN",...
```

и т. д.

Если мы не хотим, чтобы подпрограмма вывода «гуляла» по всей памяти, нам нужно уметь как-то отмечать конец текста. Самое простое решение — добавить к тексту нулевой байт и всякий раз производить проверку текущего байта на совпадение с ним:

```
LOOP:      MOV B      (R1)+,R0
            BEQ        FINIS
            .TTYOUT
            BR         LOOP
FINIS:      RTS        R5
```

Окажется ли достаточной такая последовательность команд:

```
LOOP:      .TTYOUT    (R1)+
            BNE        LOOP
            RTS        R5
```

Еще раз вернитесь к этому вопросу после прочтения гл. 4.

Единственная разница между директивами **.ASCII** и **.ASCIZ** состоит в том, что при интерпретации последней из них в конец указанного текста автоматически добавляется нулевой байт. Именно это и требуется нашей программе.

Мы могли бы вместо того, чтобы писать собственную программу, использовать и вызов монитора. В системе RT-11 это

.PRINT. Если включить в программу команду

PRINT #MEM ; обратите внимание на синтаксис!

то будет распечатана последовательность байтов, начиная с ячейки **MEM**. После обнаружения нулевого байта будет переведена строка и управление передано вызывающей программе. Все это находится в полном соответствии с директивой **.ASCIZ**. Не забывайте, что для выполнения вызова **.PRINT** в программу должна быть включена директива **.MCALL**.

Если нам нежелательно, чтобы после вызова **.PRINT** строка переводилась, нужно в конец текста поставить байт, содержащий код **200**; после обнаружения такого байта макрокоманда **.PRINT** возвратит управление вызывающей программе, не переводя при этом строку. Чтобы добавить к тексту такой байт, сначала директивой **.ASCII** заносим в память сам текст, а затем специально добавляем в его конец байт:

```
MEM:            .ASCII            /WAITING FOR GODOT/
                 .BYTE            200
```

Поскольку в результате может быть сформировано нечетное число байтов, то перед любыми дальнейшими действиями, требующими, чтобы текущий байт был четным, нужно поставить директиву **.EVEN**. Она указывает ассемблеру на необходимость пропуска одного байта, если счетчик команд содержит нечетный адрес. С четного адреса должны начинаться все команды в языке ассемблера, директивы **.WORD** и инструкции **.END**:

```
MEM:            .ASCIZ            /WAITING FOR GODOT/
                 .EVEN
                 .END            START
```

Если в этом примере директива **.ASCIZ** начинается с четного адреса, то директиву **.EVEN** включать не обязательно (так ли это?). Тем не менее быстрее и надежнее все-таки поставить эту директиву, чем считать байты.

УПРАЖНЕНИЕ. Что произойдет в результате выполнения последовательности команд

```
MEM:            .PRINT            #MEM
                 .ASCIZ            'I'm all right'
                 .END            START
```

и что нужно изменить в ней?

Локальные метки. Сложные программы, включающие большое число подпрограмм, могут служить хорошей тренировкой изобретательности программиста в создании имен. В языке

ассемблера машины PDP-11 имеются специальные «повторно-используемые» метки. Рассмотрим следующую программу ввода, в которой применяется команда сравнения байтов **CMPB** (CoMPare a Byte):

```

READ:      MOV      #MEM,R1
1$:         .TTYIN   (R1)
           CMPB     #12,(R1)+
           BNE      1$
DONE:      ...

```

(Каким образом эта программа заносит вводимый текст в память?) Выражение **1\$** (здесь \$ просто знак доллара, а не символ расширения кода **ESCAPE**) является такого рода «локальной меткой». Локальные метки должны иметь вид **1\$, 2\$, 3\$** и т. д. На локальные метки можно ссылаться точно так же, как и на «обычные метки», но только в том случае, если между строкой, содержащей локальную метку, и командой, на нее ссылающейся, *нет обычной метки*. В нашем примере метка **1\$** является локальной по отношению к части программы между метками **READ** и **DONE**. Вне их она *вообще восприниматься не будет*. Поэтому в командах, расположенных после метки **DONE** или до метки **READ** (или и там, и там), можно использовать другие метки **1\$** без боязни двусмысленности. Любая подобная ссылка интерпретируется ассемблером как ссылка на метку, расположенную в текущем локальном блоке программы, и поэтому соответствующим образом кодируется.

Учтите, что локальными метками могут быть помечены только слова, но не байты с нечетными адресами. Обычных меток подобное ограничение не касается.

3.2. Стеки

Как уже говорилось в § 3.1, *стек* есть совокупность элементов, которые заносятся в память таким способом, что записанный последним оказывается самым доступным. Стеки чрезвычайно удобны для хранения данных в вычислительной машине. Частные от деления в программе печати, а также (как мы уже видели) начальные адреса последовательно вызывающих одна другую подпрограмм являются лишь двумя из многочисленных примеров обработки информации по принципу LIFO: «последним пришел — первым обслужен» (Last In, First Out). За годы своего применения стеки получали различные названия. Часто их называют магазинами по аналогии с устройством автоматов, которые можно встретить в закусочных ¹⁾. Каждая новая тарелка

¹⁾ Часто ссылаются на аналогию с магазином винтовки. Последний вложенный в магазин патрон в ствол попадет первым, — *Прим. ред.*

«вталкивается» сверху. Чтобы взять тарелку, нужно «вытолкнуть самую верхнюю с вершины». При такой интерпретации может возникнуть превратное представление, что весь расположенный в памяти машины массив информации сдвигается, когда к нему добавляется новый элемент или удаляется элемент, записанный последним.

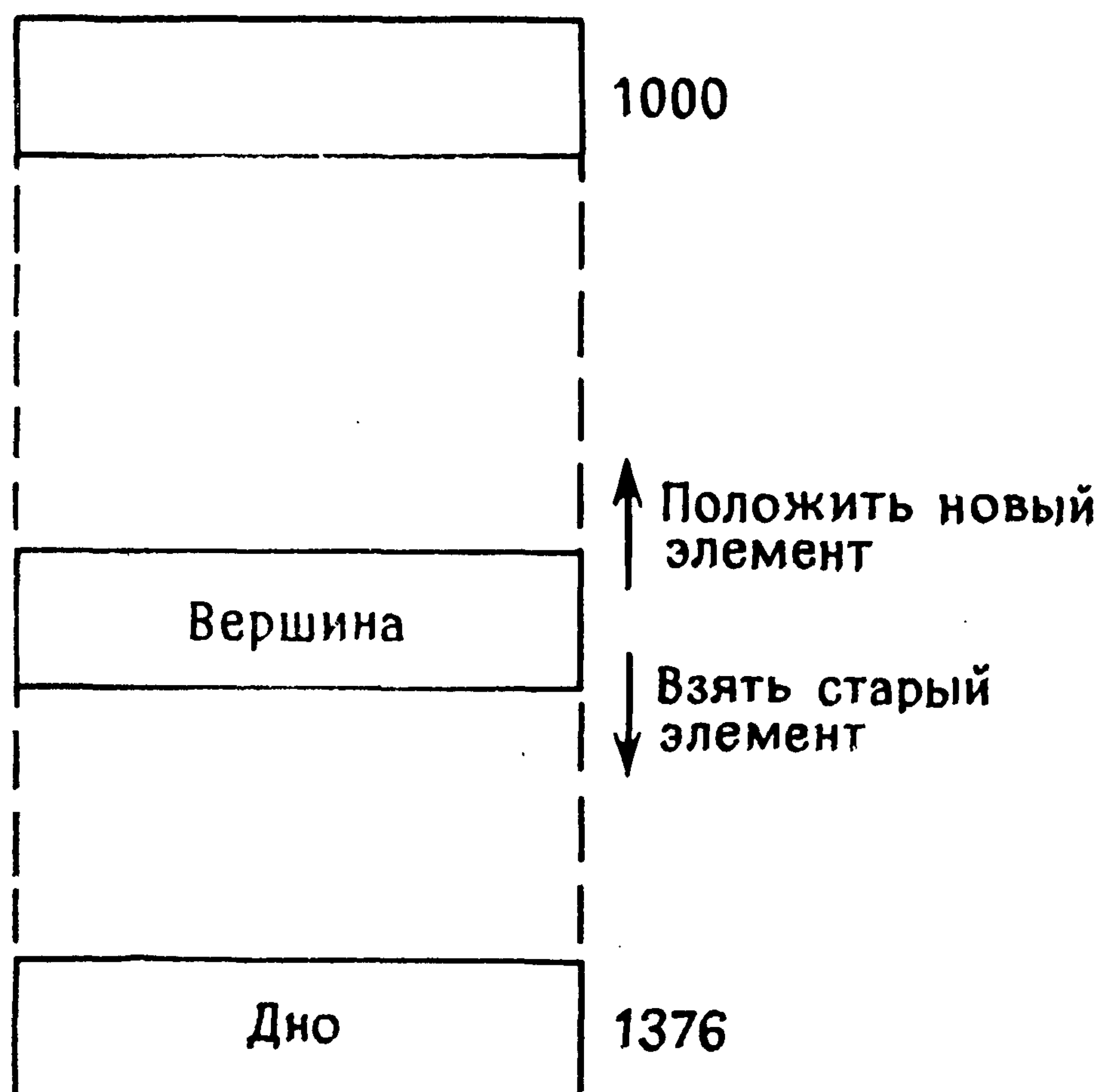
Лучше сказать, что новый элемент *кладется на* вершину стека или последний из поступивших элементов *берется с* вершины стека. На дне стека находится элемент, поступивший первым; он будет взят последним. Даже такая терминология, как мы увидим, может привести к ошибочному толкованию, хотя и не столько существенному.

Указатель стека. В системе PDP-11 стек представляет собой область памяти, отведенную программистом или операционной системой, для хранения информации по принципу LIFO. Данные, хранящиеся в стеке, выглядят точно так же, как и любые другие данные, и, более того, доступны для выборки обычным способом. Однако, чтобы блок памяти действительно стал стеком, в программе должен еще содержаться *указатель* его вершины. Занесение на вершину стека или снятие с нее осуществляется путем косвенной адресации через указатель стека.

Допустим, что в качестве указателя в программе используется регистр **R1** и что для стека отводятся ячейки с **1000** по **1400** (не включая последнюю). Это означает лишь то, что программист будет стараться избегать обращаться к этим ячейкам иначе как

через **R1**. Перед началом вычислений стек пуст. Программа должна начинаться с установки указателя стека таким образом, чтобы данные заносятся в него, начиная с ячейки, считающейся его дном.

Обычно в системе PDP-11 дном стека считается ячейка с наибольшим адресом. Как следует из примера, такое соглашение достаточно разумно. Во всяком случае его следует придерживаться по причинам, о которых сейчас будет сказано.



Поскольку элементы заносятся в ячейки стека с последовательно убывающими адресами, здесь напрашивается автодекрементный

способ адресации. Итак, сначала мы должны поставить команду **MOV # 1400, R1**, после чего содержимое ячейки **MEM** будет занесено в стек так:

MOV MEM, -(R1)

а верхний элемент будет снят со стека и занесен в ячейку **WRD** командой

MOV (R1)+, WRD

В обоих случаях регистр **R1** продолжает указывать на вершину стека.

Может показаться не совсем правильным говорить о снятии элемента со стека, поскольку команда **MOV**, выполняющая пересылку в ячейку **WRD**, ничего не изменяет в ячейке, являющейся источником данных. Все дело в том, что при снятии элемента занимаемая им ячейка не очищается, а лишь изменяется указатель стека, в результате чего она перестает быть частью стека. Если теперь в стек записывается новый элемент, то он занимает эту только что освободившуюся ячейку.

В программе удобно отводить под стек блок памяти, непосредственно предшествующий ячейке, с которой начинается ее выполнение. Так, предполагая, что наша программа должна быть загружена с тысячной ячейки, мы могли бы отвести блок памяти под стек и установить указатель следующим образом:

START:
.BLKW 400
MOV #1400, R1

Однако, чтобы завести стек, программе совсем не обязательно знать свой начальный адрес. Цель команды **MOV** состоит в том, чтобы занести в **R1** адрес ее же первого слова. Язык ассемблера системы PDP-11 воспринимает символ **.** (точка) как адрес первого слова той команды, в которой он используется. Поэтому можно заменить **MOV** командой

START: MOV #., R1

Принимая во внимание тот факт, что *ассемблер может выполнять вычисления*, не запрещается применять символ **.** и в командах перехода. К примеру, команды

TST (R0)+
BEQ .-2

будут повторяться до тех пор, пока не встретится нулевой элемент. Мы настоятельно рекомендуем избегать подобной записи. Даже удобствами ни в коей мере нельзя оправдать опасность

ошибки в вычислении величины смещения, которая особенно вероятна из-за различных длин команд в машине PDP-11. А поскольку допускаются локальные метки вида $n\$$, использование в таких ситуациях символа $.$ вообще не может иметь никаких реальных выгод.

В то же время арифметические средства ассемблера удобно использовать в выражениях типа $MEM+2$ (адрес слова, следующего за MEM), $MEM-2$ и т. п. Как вы считаете, какие проблемы могут возникнуть, если в качестве адреса указать $MEM+WRD$ или $MEM-WRD$?

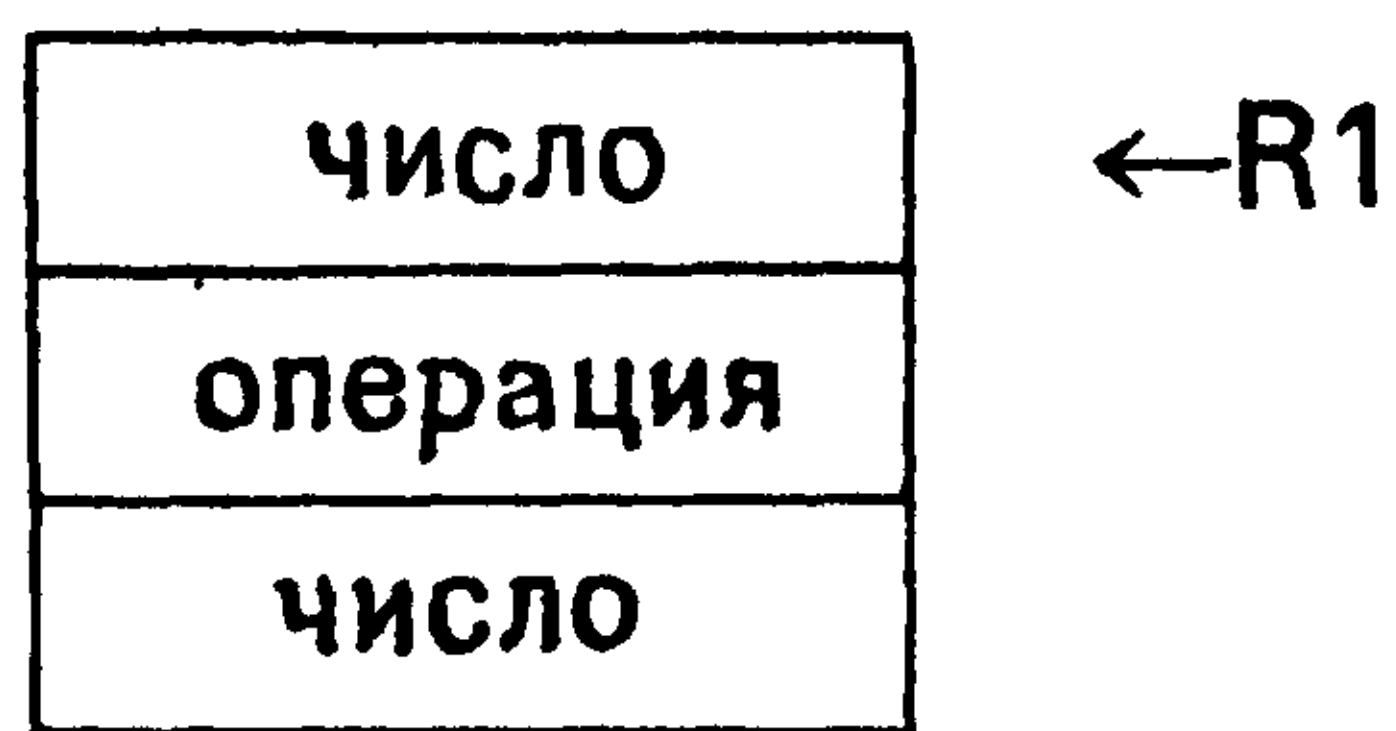
Ассемблер также может выполнять операции умножения (обозначается $*$) и деления (обозначается $/$), отбрасывая в последнем случае остаток. Учтите, однако, что вычисления арифметических выражений производятся ассемблером слева направо без учета привычного приоритета операций. Так, результат вычисления выражения $1+2*2$ равен не 5, а 6. Для задания иного порядка вычислений можно использовать *угловые скобки* $\langle . \rangle$.

Вычисление арифметических выражений. Давайте разберемся, как в ассемблере производится вычисление арифметических выражений. Поставим перед собой такую задачу: написать программу, которая читает арифметическое выражение с терминала и печатает результат его вычисления. Выражение будет представлять собой последовательность положительных чисел, чередующихся со знаками операций. Начинаться и заканчиваться оно должно числом. Вдобавок могут встречаться пары круглых скобок, причем открывающие скобки должны непосредственно предшествовать числу, а закрывающие следовать сразу после числа. Так, $3*(7-4)+1-(6-3)$ подобного рода выражение. Заметьте, что операцию умножения мы скобками не выделяем. Не забывайте также, что в выражении, включенном в написанную на языке ассемблера программу, должны применяться угловые, а не круглые скобки.

В нашей программе будет использоваться стек, а в качестве указателя стека — $R1$. Мы начнем с чтения текста в коде ASCII и записи его в блок памяти, причем $R2$ будет указывать на первое слово блока. Здесь потребуются некоторые вспомогательные подпрограммы, которые вы можете написать сами. В подпрограмме NUM будет формироваться число, первая цифра которого находится в $(R2)$. При возврате из NUM регистр $R2$ должен указывать на ячейку, следующую сразу за последней цифрой числа, а результат подпрограмма NUM заносит в стек. Подпрограмма OP должна заносить код обнаруженной по адресу $(R2)$ арифметической операции (код ASCII) на вершину стека и изменять $R2$ так, чтобы он указывал на следующую ячейку. В главной программе нужно, естественно, обеспечить, чтобы при вызове

NUM регистр **R2** указывал на первую цифру числа, а при вызове **OP** — на знак арифметической операции.

Необходимо также написать подпрограмму **CALC**, которая выполняет вычисления, когда в стеке возникает ситуация такого вида:



Подпрограмма должна выполнить требуемое действие, результат будет занесен в стек на место нижнего числа, а сами числа удалены из стека. Например, при операции **+** подпрограмма **CALC** выполнит следующее:

TST	(R1)+,(R1)+
ADD	-4(R1),(R1)

Обратите внимание на манипуляции со стеком! Заранее, конечно, неизвестно, какая операция поступит на вход подпрограммы **CALC**. Поэтому она должна начинаться с проверки содержимого ячейки **2(R1)** и дальше осуществлять переход на процедуру, выполняющую соответствующие вычисления.

УПРАЖНЕНИЕ. Напишите эти вспомогательные подпрограммы.

Ядром нашей программы будет подпрограмма **EVAL** для вычисления значения выражения. Главная программа будет считывать выражение с терминала в блок памяти по указателю **R2**, вызывать подпрограмму **EVAL**, производить вывод на печать и осуществлять выход. Фрагмент подпрограммы вывода должен напечатать содержимое ячейки, находящейся на дне стека.

Прежде всего рассмотрим, как написать подпрограмму **EVAL**, чтобы она могла работать с выражениями без круглых скобок. Мы считываем первое число и заносим его в стек. Если следующий символ есть **←|**, то вычислений не требуется, и мы возвращаемся в главную программу. В противном случае в стек заносится знак операции и следующее число. Затем вызывается подпрограмма **CALC**, чтобы выполнить требуемые вычисления, результат которых останется в стеке. Если далее следует символ **←|**, то возвращаемся в главную программу. Иначе следующий знак операции

и число заносятся в стек, и процесс повторяется.

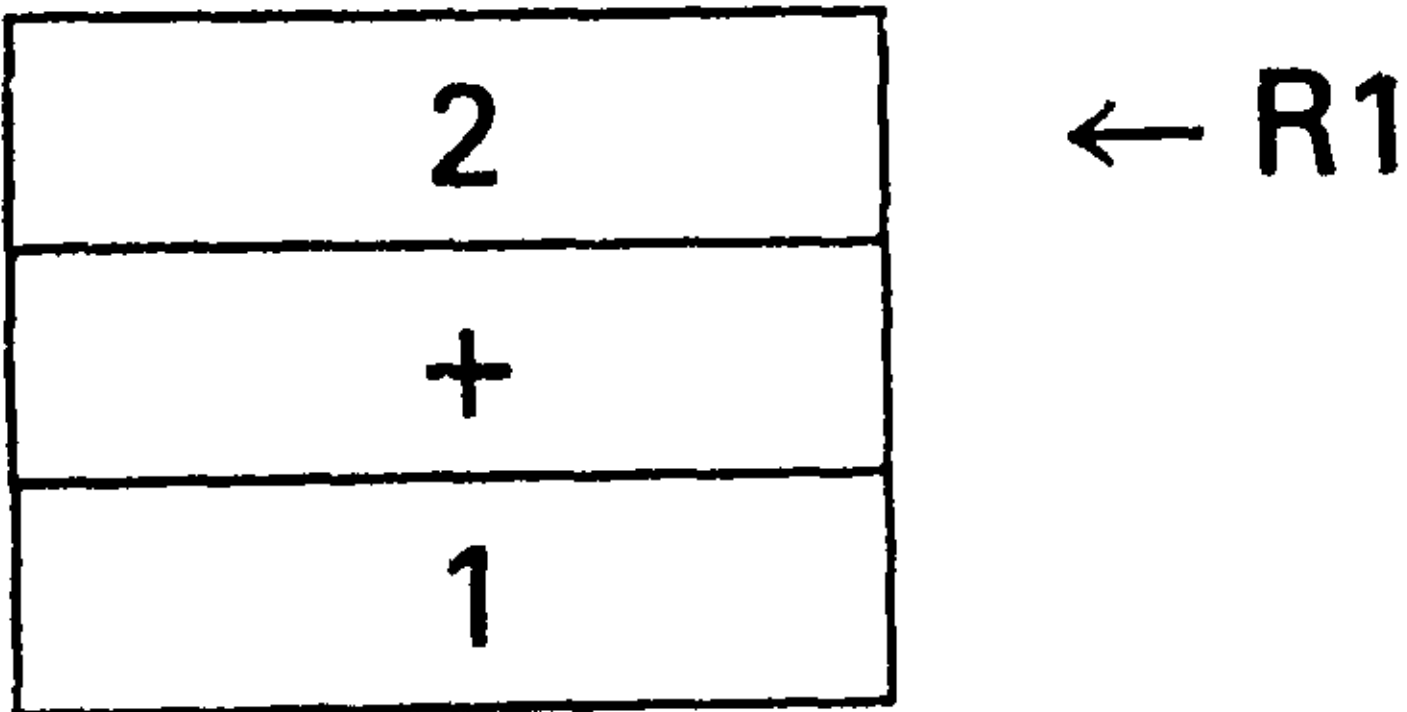
```

      EVAL:      JSR      R5,NUM
                  CMP      #15,(R2)
                  BNE      1$
                  RTS
1$:      JSR      R5,OP
                  JSR      R5,NUM
                  JSR      R5,CALC
                  CMP      #15,(R2)
                  BNE      1$
                  RTS      R5

```

- УПРАЖНЕНИЯ. 1. Сколько места под стек нужно будет отвести для такой программы?
2. Нарисуйте блок-схему программы.
3. Напишите программу и подходящим образом (где это нужно) ее прокомментируйте.

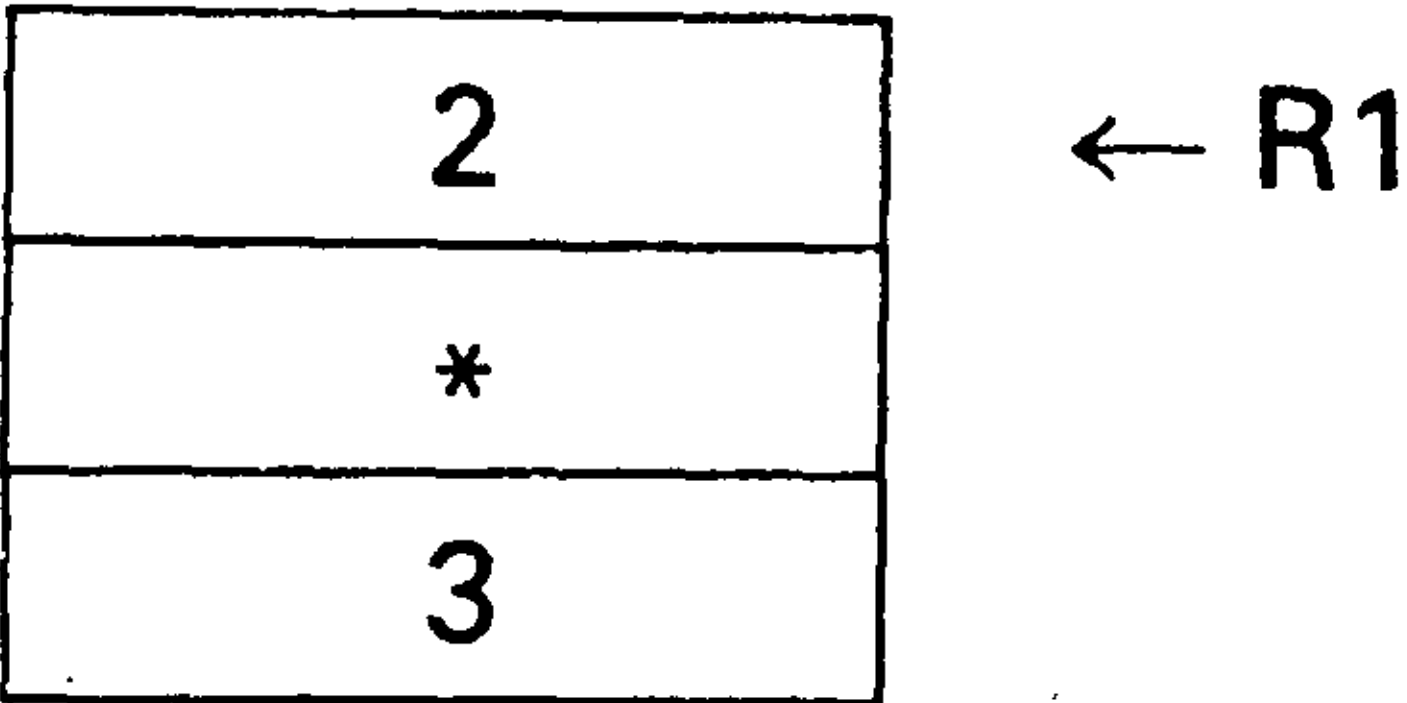
Попробуем разобраться в том, как работает наша подпрограмма. Допустим, мы набрали на терминале `1+2*2↵`. Подпрограмма **EVAL** начинается с вызова подпрограммы **NUM**, т. е. с засылки числа 1 в стек. Поскольку следующий символ не `↵`, происходит переход на метку `1$` и занесение знака `+`, а также числа 2 в стек. Теперь стек выглядит так:



Далее вызывается подпрограмма **CALC**, которая оставляет стек в таком виде:



Процесс занесения знака операции и числа повторяется, что приводит к



а после подпрограммы **CALC** в стеке остается



Следующий символ есть \leftarrow . Команда **RTS R5** возвращает нас в главную программу. Теперь печатается элемент стека (т. е. число 6) и происходит выход из программы.

УПРАЖНЕНИЕ: Измените программу так, чтобы она могла воспринимать выражения, в которых перед первым числом или после открывающей скобки стоит знак — (*унарный* минус в отличие от *бинарного*, который является знаком операции и ставится между числами).

Рекурсивные подпрограммы. Возможности нашей программы существенно расширятся, если она будет обрабатывать выражения, содержащие скобки. Посмотрим, как вычисляется, например, такое выражение: $3*(7-5)$. Сначала все происходит так же, как и в подпрограмме **EVAL**, поскольку выражение начинается с числа 3, за которым следует операция умножения. Затем встречается открывающая скобка, и мы тут же откладываем вычисление $3*$ и начинаем новое. Оно состоит в выполнении действия $7-5$. Чтобы его выполнить, мы снова проходим через подпрограмму **EVAL**, но на этот раз начальным числом становится 7. Подпрограмма **EVAL** заканчивает это подчиненное вычисление, когда встречается закрывающая скобка. Во время работы она вызывает подпрограмму **CALC**, результатом выполнения которой является число 2. Теперь можно продолжить отложенное вычисление, зная, что оно состоит в умножении $3*2$. Для этого прежняя подпрограмма **EVAL** вызывает **CALC** и потом возвращается в главную программу.

После засылки в стек первого числа и знака операции нам может встретиться как число, так и открывающая скобка. В последнем случае нужно начать еще один вычислительный процесс; другими словами, мы хотим вызвать подпрограмму **EVAL**. Подпрограмма **EVAL**, таким образом, представляет собой пример подпрограммы, обращающейся к самой себе, — *рекурсивной* подпрограммы.

Наступил подходящий момент для того, чтобы вновь обратиться к § 3.1 и проверить, не препятствует ли механизм работы команд **JSR—RTS** возможности обращения из подпрограммы к самой себе. Вам следует просмотреть рассуждения, касающиеся рис 3.2, и убедиться в том, что, даже если подпрограммы **SUB1**, **SUB2** и **SUB3** заменить одной подпрограммой **SUB**, адреса возврата останутся правильными. Конечно, при обращении рекурсивной процедуры к самой себе должны быть соблюдены некоторые условия — иначе она передаст управление неизвестно куда и никогда не возвратится назад.

Рекурсивные подпрограммы наиболее ярко демонстрируют опасность, общую для работы с любыми подпрограммами: необхо-

димо заботиться о том, чтобы вызывающая подпрограмма не портила ячейки, которые будет использовать подпрограмма вызываемая. Посмотрите, например, что произойдет, если окончательный результат работы подпрограммы **EVAL** будет помещен в нулевой регистр, а она снова вызовет саму себя. Однако то, как она была написана нами, позволяет вносить дальнейшие усовершенствования и в то же время избегать подобной опасности (поскольку для хранения промежуточных данных и конечного результата в ней используется стек, то при каждом рекурсивном вызове отводятся новые ячейки памяти).

Итог выполнения подпрограммы **EVAL** на текущем этапе ее разработки состоит в засылке перед выходом результата вычислений в стек. Если мы позаботимся о том, чтобы это свойство

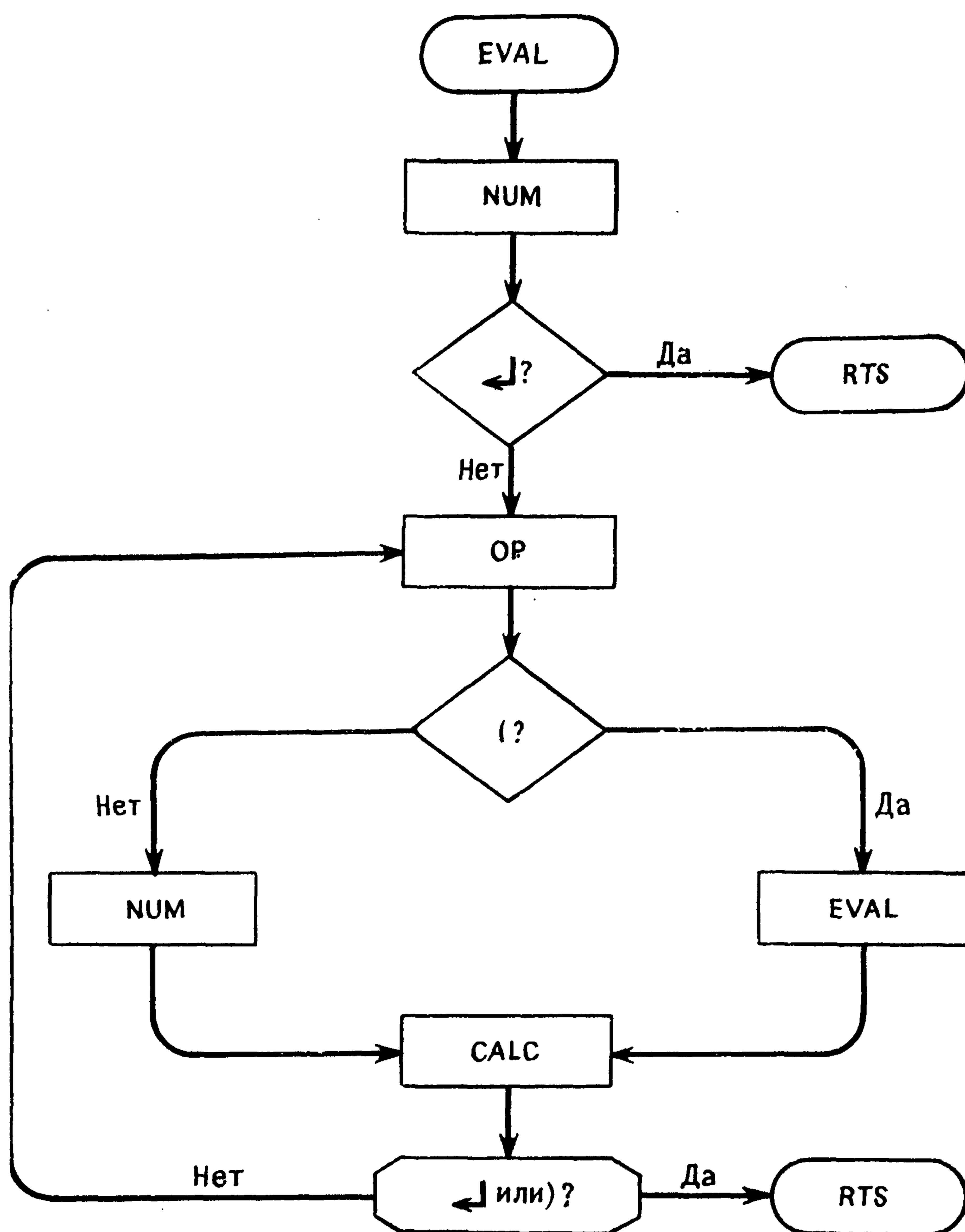


Рис. 3.3. Блок-схема для вычисления арифметических выражений.

сохранилось и в будущем, то для вызывающей программы вызов подпрограммы **EVAL** произведет действие, аналогичное вызову подпрограммы **NUM**: в стеке появится число. Значение этого

вывода состоит в том, что он позволяет сохранить очень простую структуру подпрограммы **EVAL**. Блок-схема последней представлена на рис. 3.3.

Условные переходы, как и ранее, относятся к следующему обрабатываемому символу.

УПРАЖНЕНИЕ. До сих пор мы не уделяли достаточного внимания условиям возврата из новой версии подпрограммы **EVAL**, если даже пока не учитывать, что следующий после возврата из **CALC** символ может оказаться закрывающей скобкой. Ограничивает ли это круг выражений, с которыми она может работать? В частности, может ли программа в том виде, как она представлена на рис. 3.3, вычислить выражения:

а) $(1+2)*3$?

б) $1+((2+3)+4)$?

в) $1+(2*(3+4))$?

Как нужно изменить структуру подпрограммы, чтобы снять эти ограничения?

Теперь мы должны написать подпрограмму **EVAL** так, чтобы привести ее в соответствие с блок-схемой. Текст такой программы приведен на рис. 3.4. Проследим за ее работой на приме-

EVAL:	JSR	R5, NUM	
	CMP	#15, (R2)	
	BNE	1\$	
	RTS	R5	
1\$:	JSR	R5, OP	
	CMP	#' (, (R2)	
	BNE	2\$	
	TST	(R2) +	; сдвиг последней (
	JSR	R5, EVAL	
	BR	3\$	
2\$:	JSR	R5, NUM	
3\$:	JSR	R5, CALC	
	CMP	#') , (R2) +	; сдвиг последней)
	BEQ	4\$	
	CMP	#15, -(R2)	; не), поэтому выход
	BNE	1\$	
4\$:	RTS	R5	

Рис. 3.4. Рекурсивная подпрограмма вычисления арифметических выражений.

ре выражения $3*(7-4)-(1-(6-3))$. На рис. 3.5 представлены состояния стека на различных этапах вычислений. Говоря о том или ином этапе, мы будем ссылаться на соответствующую букву внизу рисунка.

Подпрограмма **EVAL** начинает свою работу с засылки в стек числа 3 и символа *, что соответствует этапу а. После этого ей попадается символ (, в результате чего, сдвинув сначала на одну

позицию указатель **R2**, она вызывает саму себя. При этом вызове в стек последовательно заносятся символы 7, — и 4 (этап *b*), затем происходит обращение к подпрограмме **CALC**, которая вычисляет разность 7—4 и оставляет стек в состоянии *c*. Взгляните на свою подпрограмму **CALC** и убедитесь в том, что именно в этом и состо-

						3				
						—				
		4				6	3			
		—				—	—			
		7	3			1	1	1	—2	
×	×	×		—	—	—	—	—	—	
3	3	3	11	11	11	11	11	11	11	13
<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>	<i>j</i>	

Рис. 3.5. Состояния стека в процессе вычисления выражения $3 \times (7 - 4) - (1 - (6 - 3))$.

ит ее функция. Подпрограмма **EVAL** встречает теперь символ **)** и осуществляет возврат. Обратите внимание на то, как в командах **CMR**, стоящих после метки 3\$, применяется автоматическое изменение содержимого регистра, позволяющее на выходе обеспечить, чтобы **R2** не указывал на открывающую скобку и в то же время чтобы не был пропущен символ **←|**. Возврат происходит на команду, следующую после **JSR R5, EVAL** в подпрограмме **EVAL**, — ведь выполнение «главной» **EVAL** еще не закончено. И из листинга программы, и из блок-схемы ясно, что теперь **EVAL** вызывает **CALC**, которая выполняет операцию 3×3 и оставляет стек в состоянии *d* (восьмеричное представление). В «главной» подпрограмме **EVAL** теперь управление передается обратно на метку 1\$ и заносится в стек символ **—**, что соответствует этапу *e*. Потом, встретив открывающую скобку, она вызывает себя. Эта «вспомогательная» **EVAL** заносит 1 и **—** в стек (этап *f*), наталкивается на новую открывающую скобку и снова обращается к самой себе. Эта «еще более вспомогательная» **EVAL** заносит в стек 6, **—** и 3, что соответствует этапу *g*; вызывает **CALC**, которая оставляет стек в состоянии *h*, и возвращается на строчку в подпрограмме **EVAL**, следующую за командой **JSR R5, EVAL**. Адреса возврата из подпрограмм хранятся во внешнем по отношению к ним стеке, и этот выход осуществляется в предыдущую («вспомогательную») **EVAL**. «Вспомогательная» **EVAL** вызывает теперь **CALC**, после выполнения которой стек находится в состоянии *i*; затем, встретив закрывающую скобку, она передает управление главной подпрограмме **EVAL**. Последняя вызывает **CALC**, после чего стек переходит в состояние *j*, и по завершающему символу **←|** осуществляется возврат в главную программу, которая теперь печатает результат.

- УПРАЖНЕНИЯ. 1. Напишите полностью всю программу.
2. Измените ее так, чтобы выражение могло начинаться с открывающей скобки, о чем уже упоминалось выше.
3. Измените программу, чтобы в выражении можно было употреблять унарный минус.
- 4*. Измените программу так, чтобы (как это обычно принято) до тех пор, пока с помощью скобок не будет указано противное, приоритет операций $*$ и $/$ был бы выше приоритета операций $+$ и $-$.
5. Вставьте в программу фрагмент печати сообщений об ошибках, если в обрабатываемом выражении не согласовано число открывающих и закрывающих скобок.

Системный стек. Из описания работы команд **JSR** и **RTS** вы уже, наверное, догадались, что аппаратура системы PDP-11 пользуется стеком, причем в качестве его указателя всегда берет-ся регистр номер 6. Этот регистр называется *аппаратным указателем стека*, и ему, как правило, дается имя **SP**. В программе обычно следует объявить

SP = %6

Теперь мы знаем, что команда

JSR R5,SUB

эквивалентна последовательности

MOV R5, -(SP)
MOV "return PC", R5
MOV #SUB, PC

в то время как **RTS R5** — последовательности

MOV R5, PC
MOV (SP) +, R5

Помните, однако, что множество действий этих команд выполняется аппаратурой как одна операция. Их нельзя «транслировать» в эквивалентные им последовательности, и последние нужно рассматривать лишь как описание, но не как определение.

Любая операционная система отводит место под стек и устанавливает **SP**. Обычно системный стек начинается с ячейки 776, и отводимое под него пространство простирается до ячейки 400. Программы пользователя загружаются сразу после области стека, начиная с ячейки 1000.

Если программа выполняется на изолированной машине без операционной системы, которая подготавливает системный стек,

то программу следует начинать с команды

MOV #..SP

Нужно также позаботиться о том, чтобы перед программой оставалось достаточно свободного места под стек. Об этом будет сказано в конце данного параграфа.

В тексте программы к **SP** можно обращаться так же, как и к другим регистрам, за исключением того, что аналогично **PC** содержимое **SP** всегда должно быть четным, поскольку должно указывать на адрес слова. В частности, системный стек можно использовать в программах в качестве их собственного стека. Это действительно хорошая мысль: если **SP** и **PC** имеют свои особые функции, а нужен еще регистр связи между подпрограммами, то программе совсем не просто пожертвовать под стек один из пяти оставшихся регистров. В итоге адреса возврата из подпрограмм и данные будут храниться в одном и том же стеке. Обычно это обеспечивает большую гибкость при программировании, хотя иногда из-за боязни запутаться используют отдельный программный стек.

УПРАЖНЕНИЕ. Измените программу **EVAL** так, чтобы в ней вместо **R1** в качестве указателя стека использовался **SP**. Проанализируйте рассмотренный выше алгоритм вычисления выражения и приготовьте таблицу состояний стека, подобную приведенной на рис. 3.5. Как вы считаете, улучшилась ли от этого программа?

Проницательный читатель может резонно спросить, так уж ли нам необходим особый регистр для осуществления обращений к подпрограммам, если мы решили экономить на использовании регистров? Почему бы при вызове подпрограммы не занести адрес возврата в стек, а при выходе из нее не взять его оттуда? Такое решение действительно возможно, и достигается оно путем обращения к **PC** как к регистру связи при вызове подпрограммы

JSR PC,SUB

и при выходе из нее

RTS PC

В § 3.4 мы покажем преимущества заведения особого регистра связи.

Сопрограммы. Эта тема слишком сложна, чтобы обстоятельно здесь ею заниматься, но реализация сопрограмм в PDP-11 настолько изящна, что о ней нельзя умолчать.

Мы видели, что если в команде **JSR** в качестве регистра связи выступает счетчик команд, то она загружает исполнительный адрес в **PC**, а адрес возврата в системный стек. Интересная форма команды получается, если адрес программы, которой должно быть передано управление, сам был занесен в стек. Заметим, что в этом случае требуемый исполнительный адрес есть не (**SP**) (т.е. ячейка в системном стеке), а ячейка, на которую указывает (**SP**). Поэтому управление передается командой

JSR PC,@(SP)

Это приводит к засылке адреса возврата в стек поверх того адреса, на который только что было передано управление. Нет, однако, никакой причины, ради которой последний должен оставаться в стеке. Лучше, прежде чем заносить текущее значение **PC**, его оттуда убрать. Это достигается таким видоизменением команды:

JSR PC,@(SP)+

что приводит к обмену содержимого **PC** и верхней ячейки стека. (Какой порядок операций внутри команды **JSR** при этом подразумевается?) Из симметрии следует, что программа, которой было передано управление, может вернуть его назад точно такой же командой, после чего в стеке будет содержаться адрес следующей за ней ячейки. Новая команда **JSR PC,@(SP)+** передаст управление именно на эту ячейку.

У нас сложилась ситуация, когда две программы могут поочередно передавать управление друг другу. Такие программы называются сопрограммами, что подчеркивает факт их кооперации, основанной на равноправии, а не на подчинении, как в подпрограммах.

Сопрограммы оказываются незаменимыми, когда программа разделяется на два основных задания, которые должны быть между собой скоординированы, но при этом их слишком тесная взаимосвязь ведет к излишним усложнениям. Программа, с задания которой начинается работа, занесет адрес второй ячейки в стек. В дальнейшем, как только одной программе требуется помощь другой, она вызывает ее командой **JSR PC,@(SP)+**. Системные программы часто выполняют столь переплетенные функции, что в них приходится использовать сопрограммы. На рис. 3.6 показана упрощенная схема взаимодействия двух процессов — чтения и интерпретации, которые протекают внутри ассемблера, когда он расшифровывает мнемонику очередной команды.

Счетчик адресов. Точка . представляет собой символ для обозначения *счетчика адресов*, который заведен внутри программы ассемблера и увеличивается в процессе трансляции программы.

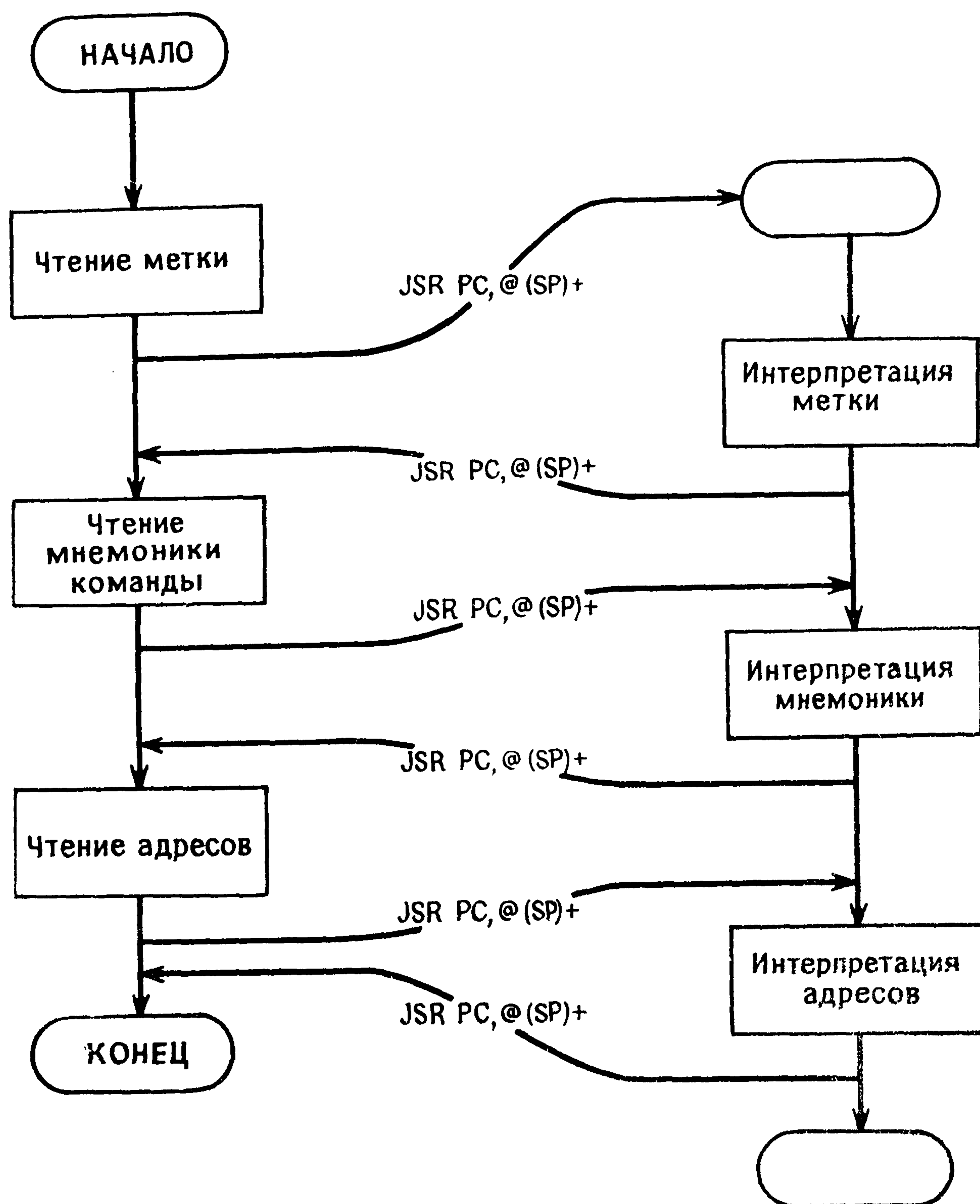


Рис. 3.6. Передача управления между сопрограммами в ассемблере.

Мы встретились с использованием точки для обозначения адреса первого слова той команды, операндом которой она является. Так, для изначальной подготовки системного стека достаточно команды

MOV #.,SP

Это действие, однако, бессмысленно, если перед программой не зарезервировано место в памяти. Большинство операционных систем гарантирует отведение под стек определенного количества памяти, загружая программу с тысячного адреса. Если требуется еще больший, чем автоматически предоставляемый системой, объем памяти, то в начале программы можно поставить директиву **.BLKW** и тем самым увеличить начальный адрес, т. е.

адрес первой выполняемой команды. Директива **.BLKW** заставляет транслятор увеличить счетчик адресов на удвоенную величину параметра (счетчик адресов подсчитывает байты). Альтернативой директиве **.BLKW** является директива **.BLKB**, служащая для резервирования блока байтов. Поэтому команды

.BLKW	40
.BLKB	100

эквивалентны друг другу. Того же результата можно достичь другим способом — изменением значения счетчика адресов оператором прямого присваивания:

. = . + 100

Начальный загрузочный адрес программы можно также задать явно, установив требуемое значение счетчика адресов оператором прямого присваивания. Например, может понадобиться, чтобы программа загружалась с ячейки **1400**. В большинстве случаев, однако, мы не можем начать программу с оператора

. = 1400

потому что почти все ассемблеры, если не объявлено противное, рассматривают любую программу как перемещаемую. Поэтому значение счетчика адресов также величина перемещаемая и в начале трансляции равна перемещаемому нулю. Окончательное значение перемещаемого нуля будет зависеть от компоновщика; во время трансляции нельзя зафиксировать привязку к адресу **1000**, и любую подобную попытку ассемблер будет считать ошибкой.

Все же можно дать указание ассемблеру производить трансляцию в абсолютных адресах директивой **.ASECT**. После нее точка **.** также воспринимается в абсолютном смысле и может быть приравнена любому требуемому адресу: **. = 1400**.

В следующих параграфах мы увидим, что при программировании может понадобиться заполнять ячейки, расположенные в начале памяти. Они служат для выполнения различных аппаратных функций. Пусть, к примеру, двенадцатая ячейка должна содержать код **340**, а в остальном программа должна остаться обычной перемещаемой программой. Вместо того чтобы устанавливать значение этой ячейки в процессе исполнения (каким образом?), оно может быть занесено во время трансляции:

.ASECT	
. = 12	
.WORD	340

Потом директивой **.CSECT** можем указать ассемблеру, чтобы все последующие команды воспринимались им как перемещаемые, и продолжить нашу программу.

Программа может включать несколько директив **.ASECT** и **.CSECT**, каждая из которых открывает абсолютную или относительную секцию. В ассемблере заведены два отдельных счетчика для двух видов команд, поэтому новая относительная секция будет автоматически загружена сразу после предыдущей.

УПРАЖНЕНИЯ. 1. Какой способ заполнения ячейки 12 вам больше нравится?

2. Что делает команда **MOV ., SP**?

3. Что общего между **.** и **PC**?

4. Пусть команда **MOV #., SP** загружается, начиная с ячейки 1400. Каким будет код этой команды? Какой код для нее сформировал ассемблер? Зависят ли ваши ответы от того, где расположена команда: внутри абсолютной или относительной секции?

3.3. Управление выполнением программы

В структурированной программе, состоящей из некоторого числа подпрограмм, должным образом скоординированных между собой для решения поставленной задачи, на различных этапах исполнения приходится принимать решения о том, куда следует передать управление. Единственная информация, на основе которой могут быть приняты подобные решения,— это содержимое регистров, ячеек памяти и состояние условных признаков.

Передача управления будет осуществлена командами условных переходов, но программист должен позаботиться о представлении информации, получаемой при вычислении, в подходящей для соответствующей команды перехода форме. Например, дальнейшее выполнение программы может зависеть от того, четное или нечетное число находится в ячейке **MEM**. Однако команды, выполняющей именно такую проверку, нет. Поэтому мы должны извлечь информацию из ячейки **MEM** в форме, которая приемлема для какой-нибудь имеющейся команды. Так, команды **BEQ** и **BNE** различают, равен операнд нулю или нет, проверяя состояние бита **Z**. Поэтому если мы занесем куда-нибудь нуль, когда значение **MEM** четно, и не нуль, когда оно нечетно (или наоборот), то наша задача будет решена. Можно, очевидно, разделить содержимое **MEM** на два и проверить остаток.

Логические команды. Мы только что решили задачу определения четности числа, но очень уж топорно. Пока мы не знаем лучшего способа, но и так ясно, что бóльшая часть вычислительных затрат пропадет впустую: нас не интересует не только част-

ное, но и точное значение остатка. Метод, при котором основная часть вычисленных значений не представляет интереса, очевидно, не лучший путь программирования.

Чтобы найти более удовлетворительное решение, мы должны освободиться от привычки считать, что ЭВМ видит в числе то же самое, что и мы. Именно такой взгляд послужил причиной *арифметического* подхода (деление содержимого **МЕМ** на два).

Содержимое ячейки **МЕМ** определяется состоянием битов, и возникает законный вопрос: как выглядит совокупность битов, если она обозначает четное число? Вы уже наверняка сообразили, что в кодах четных чисел нулевой разряд содержит 0, а в кодах нечетных 1. (Убедитесь, что для отрицательных чисел это тоже верно.) Итак, мы свели нашу задачу к проверке нулевого разряда.

Команды, предназначенные для работы со словами, содержимое которых рассматривается как шкала битов (а не число), называются *логическими* (в отличие от арифметических) командами. Такая терминология связана с тем, что если интерпретировать 1 в разряде как «истину», а 0 как «ложь», то содержимое слова можно рассматривать как набор логических значений. Поэтому указанные команды выполняют классические логические операции. Так, например, если p и q представляют собой выражения, то в логике выражение « p и q » (символически записываемое в виде $p \wedge q$) истинно лишь тогда, когда и p , и q истинны.

Команда проверки битов **BIT** (BIt Test), входящая в систему команд машины PDP-11, осуществляет логическую функцию *и* над своими операндами. Так,

BIT MEM,WRD

формирует **(MEM) \wedge (WRD)**, т. е. слово, в котором бит равен 1 в том и только том случае, если в обоих операндах **(MEM)** и **(WRD)** соответствующие биты равны 1. Поэтому, если в **R0** записать 1, то результатом выполнения команды **BIT R0,MEM** будет 0, если **(MEM)** четно, и 1 в противном случае.

Команда **BIT** формирует результат в недоступном для программы внутреннем регистре, не затрагивая при этом содержимое своих операндов. Условные признаки устанавливаются в соответствии со значением результата. Таким образом, команда **BIT** есть логический аналог команды **СМР**. Поскольку в ней разрешены любые методы адресации, проверка на четность может быть такой:

	BIT	#1,MEM
	BEQ	EVEN
ODD:	...	

Так как любой разряд может представлять «истину» или «ложь», то на самом деле команда **BIT** одновременно выполняет шестнадцать логических операций. Незнакомый с формальной логикой читатель может рассматривать логические операции просто как команды, в результате выполнения которых получаются те или иные конфигурации битов. Все же на некотором этапе полезно ознакомиться с элементами логики. Это особенно относится к программистам, пользующимся языками высокого уровня типа Фортран или Паскаль, в которых логические операторы (такие, как **IF**) находят широкое применение.

Чтобы определить результат логической операции, мы должны знать, как она выполняется при всевозможных комбинациях битов источника и приемника. Для каждого бита нужно рассмотреть четыре возможности: биты в источнике и приемнике равны 0; оба бита равны 1; бит в источнике равен 1, а в приемнике 0; в источнике 0, в приемнике 1. В связи с этим подобную операцию можно задать таблицей конфигураций. Для команды **BIT** она имеет вид

	Источник	0	0	1	1
	Приемник	0	1	0	1
BIT		0	0	0	1

УПРАЖНЕНИЯ. 1. Каково состояние разрядов **Z** и **N** после выполнения команды **BIT MEM, WRD**, если содержимое **MEM** и **WRD** соответственно равно:

- а) 100, 200?
- б) 100, 300?
- в) 177777, — 77777?

2. Как бы вы стали проверять, делится ли содержимое ячейки **MEM** на четыре?

Другими основными логическими операциями, помимо операции *и*, являются *или* и *не*. Операция *или* требует аккуратности, потому что не всегда бывает ясно, имеется ли в виду *включающее или*, обозначаемое символом \vee и определяемое так:

« $p \vee q$ » истинно, если p , или q , или оба истинны;

или подразумевается *исключающее или* (обозначаемое ∇), которое определяется по-другому:

« $p \nabla q$ » истинно, когда p или q , но не оба одновременно истинны.

Задающие эти операции таблицы могут быть объединены в одну:

	0	0	1	1
	0	1	0	1
\vee	0	1	1	1
∇	0	1	1	0

На машине PDP-11 операцию *включающее или* производит команда установки битов **BIS** (BIt Set). В отличие от команды **BIT** она заносит результат по адресу приемника. Так, команда

BIS #20, MEM

установит четвертый разряд ячейки **MEM** независимо от его первоначального состояния.

Исключающее или выполняется командой **XOR**¹⁾ синтаксис которой особый:

XOR R, X

Здесь **R** — регистр, **X** — адрес приемника. Над содержимым **R** и **X** производится операция *исключающее или*, и ее результат заносится в **X**. Например, после выполнения команды

MOV #20, R0
XOR R0, MEM

в четвертом бите ячейки **MEM** будет 0, если в исходном состоянии там была 1, и будет 1, если до операции там был 0.

Логическая операция *не*, обозначаемая \neg (или \sim), служит для получения *логического дополнения* к содержимому данного слова: все 1 становятся 0 и наоборот. Она выполняется одноадресной командой **COM** (COMplement).

Последняя логическая команда — очистка битов **BIC** (BIt Clear):

BIC MEM, WRD

Она формирует в ячейке **WRD** следующую логическую функцию: $\{\neg \text{MEM}\} \wedge (\text{WRD})$. То есть бит в **WRD** становится нулевым, если соответствующий ему бит в **MEM** равен 1. Например, после выполнения команды

BIC #20, MEM

четвертый бит в ячейке **MEM** будет равен 0.

Все рассмотренные команды заносят 1 или 0 в биты **N** и **Z** в зависимости от значения результата.

¹⁾ Отсутствует на некоторых небольших процессорах.

УПРАЖНЕНИЯ. 1. Напишите программу, осуществляющую функцию *и* над двумя своими операндами.

2. Какая команда меняет код ASCII для знака $+$ на код для знака $-$, и наоборот?

3. Всегда ли одной логической командой можно произвести взаимную замену кодов ASCII для любых двух заданных литер?

4. Чем отличаются команды **COM** и **NEG**?

5*. Напишите программу, которая будет по очереди запрашивать ввод: мнемоники команд **BIT**, **BIS**, **XOR** или **COM**; содержимого источника; содержимого приемника. Затем программа должна напечатать содержимое приемника после операции, а также состояние битов *N* и *Z*.

Переполнение. Машина PDP-11, как мы ее описывали до сих пор, выполняет два существенно различных типа команд: логические, в которых слово рассматривается просто как совокупность шестнадцати битов, и арифметические, в которых считается, что слово содержит один знаковый бит плюс пятнадцать битов, определяющих величину числа.

В действительности ЦП PDP-11 — более гибкое устройство. В командах **ADD** и **SUB** не учитывается разница между 15-м разрядом и остальными. Однако в аппаратуре заложены возможности, которые программист может использовать для получения правильного знакового разряда и, таким образом, корректно выполнить арифметическую операцию.

Предположим, что нам нужно сложить с самим собой число $2^{14} + 1$. Оно кодируется нулем в 15-м разряде (поскольку положительно), единицами в разрядах 14 и 0 и нулями во всех остальных. Сложение выполняется следующим образом:

$$\begin{array}{r} 010 \dots 001 \\ + 010 \dots 001 \\ \hline 100 \dots 010 \end{array}$$

С точки зрения двоичной арифметики все выглядит правильно до тех пор, пока мы не вспомним, что пятнадцатый разряд знаковый. В нижней строке вместо правильного результата $2^{15} + 2$ оказалось число $-2^{15} + 2$. Заметьте, что полученное число не есть правильный результат с отрицательным знаком. В то же время, если мы будем рассматривать нижнюю строку как шестнадцатиразрядное положительное число, она дает правильный ответ.

Ситуация, когда абсолютная величина результата арифметической операции просачивается в знаковый разряд, называется *переполнением*. Ясно, что оно может возникнуть и при вычитании, например при выполнении команды **SUB MEM, WRD**, где **MEM** содержит $-(2^{14} + 1)$, а **WRD** содержит $2^{14} + 1$.

Переполнением иногда называют также и несколько иную ситуацию. Допустим, что нам нужно выполнить команду **ADD MEM, MEM**, причем **MEM** содержит -1 . Следовательно, все биты в ячейке **MEM** равны 1, и мы имеем

$$\begin{array}{r} 111 \dots 111 \\ + 111 \dots 111 \\ \hline 1111 \dots 110 \end{array}$$

Конечно, результат не помещается в рамки слова машины PDP-11. Сложение двух шестнадцатиразрядных двоичных чисел с единицей в старшем разряде привело к семнадцатиразрядному результату. Однако для самого левого в приведенном примере добавочного разряда нет места в машинном слове; следовательно, он должен быть отброшен. Позже мы обсудим, что с ним происходит, но *ничего общего с переполнением подобная ситуация не имеет*. Когда мы его отбрасываем, то получается шестнадцатиразрядное число с нулевым правым и единичными остальными битами. Это число -2 , представленное в дополнительном коде, как и должно быть. Заметьте, что сложение производилось над шестнадцатиразрядными числами без особого выделения знакового разряда, и все же результат оказался правильным. Это не переполнение.

Рассмотрим, однако, попытку выполнить сложение **ADD MEM, MEM**, если **MEM** содержит $-2^{15} + 1$. В таком числе биты 15 и 0 равны 1, а остальные — нули.

$$\begin{array}{r} 100 \dots 001 \\ + 100 \dots 001 \\ \hline 1000 \dots 010 \end{array}$$

Если мы теперь отбросим левый бит, то получим 2, в то время как правильный результат $-2^{16} + 2$. Подобная ситуация называется переполнением *не* потому, что левый разряд отбрасывается, а потому, что арифметическая операция не дает результата с правильным знаком. Величина результата также получается неверной, но это уже следствие переполнения, не имеющее отношения к его определению. Итак, переполнение случается тогда, когда знак результата арифметической операции неправильный.

УПРАЖНЕНИЯ. 1. Может ли второй тип переполнения (причем, когда результат положителен) случиться при вычитании?

2. Если сложение двух чисел m и n (положительных или отрицательных) вызывает переполнение, то будет ли оно происходить при сложении чисел $-m$ и $-n$?

ЦП снабжен схемой распознавания случая переполнения и реагирует на него установкой первого бита PS (бит V).

Арифметические команды, такие, как **ADD**, **SUB**, **CMR**, **NEG**, **DEC** или **INC**, способные вызвать переполнение, очищают бит

V, если оно не возникает. Команды **MOV**, **CLR** и логические команды всегда очищают бит V.

УПРАЖНЕНИЕ. Как вы считаете, если **R0** содержит число 177776, то будет ли установлен бит V в результате выполнения команды **TST (R0)+**?

Имеются команды перехода, реагирующие на состояние бита V; команда перехода, если бит V равен 1, **BVS** (Branch if V is Set), и команда перехода, если бит V равен 0, **BVC** (Branch if V is Clear).

Допустим теперь, что нам нужно в программе организовать условный переход в зависимости от результата сравнения содержимого ячеек **MEM** и **WRD**. Рассматривая их как обычные числа, которые могут быть положительными или отрицательными, нам нужно перейти на метку **ONWARD**, если $(MEM) \geq (WRD)$. Легко убедиться в том, что наш предыдущий подход

CMP	MEM,WRD
BPL	ONWARD

приведет к ошибке, если операция вычитания, осуществляемая командой **CMP**, даст переполнение. Так, если **MEM** содержит 2^{14} , а **WRD** содержит -2^{14} , то, безусловно, $(MEM) \geq (WRD)$. Однако команда **CMP MEM,WRD** из-за переполнения сформирует результат с 1 в пятнадцатом бите и нулями во всех остальных. Команда **BPL** не сможет распознать, что мы имеем в виду операцию $2^{14} - (-2^{14}) = 2^{14} + 2^{14} = 2^{15}$, поскольку функция **BPL** ограничена проверкой бита N. В нашем случае бит N равен 1, так как он устанавливается всегда, когда результат вычислений имеет 1 в старшем разряде (это справедливо и для логических команд). Итак, команда **CMP** установит бит N, и, следовательно, команда **BPL** не приведет к ветвлению.

Поэтому, когда в результате выполнения команды **CMP** бит N равен 1, прежде чем осуществить переход, необходимо проверить, что 1 в пятнадцатом разряде результата есть следствие переполнения, а не признак «отрицательности» числа. Иными словами, если оба бита N и V равны 1, то все же мы должны осуществить переход:

CMP	MEM,WRD
BPL	ONWARD
BVS	ONWARD

Обратите внимание на то, что, поскольку команды перехода не меняют условных признаков, одной команды **CMP** вполне достаточно.

К сожалению, это все еще не приведет к тому, что мы хотели, поскольку здесь не принимается во внимание второй тип переполнения. Пусть **MEM** содержит -2^{15} , в то время как **WRD** содержит $2^{15}-1$. Поскольку **(MEM)** отрицательно, а **(WRD)** положительно, мы, естественно, имеем **(MEM) < (WRD)**, и потому перехода не должно быть. Но вычитание выполняется так:

$$\begin{array}{r} 100 \dots 000 \\ -011 \dots 111 \\ \hline 000 \dots 001 \end{array}$$

Поскольку пятнадцатый бит нулевой, результат «выглядит» положительным, и ЦП сбросит бит **N**. Однако арифметическая операция привела к результату с неверным знаком: он должен был быть отрицательным, так как $-2^{15} - (2^{15}-1) = -2^{15} - 2^{15} + 1 = -2^{16} + 1$. Поэтому ЦП установит бит **V**, и наши команды вызовут переход.

Таким образом, когда бит **N** нулевой, перед тем как осуществлять переход, необходимо проверить, что **0** в пятнадцатом бите результата говорит о переполнении, а не о «положительности» полученного числа. Иными словами, когда бит **N** нулевой, а бит **V** равен **1**, нам *не следует* производить переход. Итак, вывод таков: вместо первоначального варианта с командой **BPL** мы должны осуществлять переход, если одновременно оба бита **N** и **V** либо **0**, либо **1**, и воздержаться от перехода, если один из них **1**, а другой **0**. На языке логики условие перехода записывается так: $N \nabla V = 0$. В точности такое условие реализуется командой **BGE** — переход, если больше или равно (Branch if Greater than or Equal).

CMP	MEM,WRD
BGE	ONWARD

УПРАЖНЕНИЯ. 1. Запишите предыдущее условие перехода, используя только команды **BPL**, **BMI**, **BVS** и **BVC**.

2*. Не применяя команд, которые проверяют значение бита **V**, напишите такую подпрограмму **VSET**, что при обращении **JSR PC,VSET** в регистр **R0** будет записана единица тогда и только тогда, когда непосредственно предшествующая этому обращению команда устанавливает бит **V**.

Дополнительной к команде **BGE** является команда **BLT** — переход, если меньше (Branch if Less Than). То есть переход по команде **BLT** происходит в том случае, когда один из битов **N** или **V** установлен: $N \nabla V = 1$. Имеется также команда **BGT** — переход, если больше (Branch if Greater Than), по которой переход осуществляется, как и по команде **BGE**, но только при этом бит **Z** не должен быть установлен: $Z \vee (N \nabla V) = 0$ — условие пере-

хода. Дополнительная к ней — команда **BLE** — переход, если меньше или равно (Branch if Less or Equal).

Перенос. Мы выяснили, что сложение двух шестнадцатиразрядных чисел со старшим разрядом, равным 1, приводит к семнадцатиразрядному результату, старший разряд которого также равен 1, и что для этого разряда не хватает места в машинном слове PDP-11. Когда 1 переходит за левый край слова, говорят, что возникает *перенос* за пределы слова. ЦП реагирует на такую ситуацию установкой бита **C**, который представляет собой нулевой бит **PS**.

Логические команды, а также команда **MOV** не влияют на бит **C**, причем заметьте, что они всегда сбрасывают бит **V**. Команда **TST** сбрасывает бит **C** так же, впрочем, как и бит **V**. Команды **INC** и **DEC** не изменяют состояние бита **C** (как они воздействуют на бит **V**?).

Команда **ADD** устанавливает бит **C**, если в результате операции возникает перенос за пределы слова, как это только что было описано; в противном случае она его сбрасывает.

УПРАЖНЕНИЯ. 1. Как изменяют состояние бита **C** команды **CLR**, **COM** и **NEG**?

2. Покажите на примерах, что по состоянию бита **C** после выполнения команды **ADD** невозможно определить, был ли получен правильный результат.

Команды **SUB** и **CMR** устанавливают бит **C**, когда при вычитании необходим перенос единицы в левый бит слова; в противном случае они его сбрасывают. Так, рассмотрим выполнение вычитания 5—7:

$$\begin{array}{r} 000 \dots 101 \\ - 000 \dots 111 \\ \hline 111 \dots 110 \end{array}$$

Чтобы получить результат, в левый разряд двоичного представления числа 5 должна быть перенесена единица. Представление числа —2 в дополнительном коде содержит 0 в правом разряде и 1 в стольких разрядах, сколько их вмещает машинное слово. Поэтому, если бы слева от слова был еще один дополнительный бит, то его нужно было бы установить в 1. Следовательно, ЦП вправе считать, что за левым краем слова находится 1, и поэтому установить бит **C**. Заметьте, что с арифметической точки зрения результат корректен, и потому бит **V** сбрасывается.

УПРАЖНЕНИЯ. 1. Покажите на примерах, что по состоянию бита **C** после выполнения команды **SUB** нельзя судить о корректности полученного результата.

2. Как могут быть записаны условия установки бита *C* в командах **SUB** или **CMR**, если вычитание рассматривать как прибавление числа в дополнительном коде?

Имеются команды, реагирующие на состояние бита *C*: команда **BCS** перехода, если бит *C* установлен (Branch if *C* is Set) и команда **BCC** перехода, если он сброшен (Branch if *C* is Clear). Эти команды имеют альтернативную мнемонику: **BCS** есть то же, что и **BLO** (Branch if **LO**wer — переход, если ниже), а **BCC** — то же, что и **BHIS** (Branch if **HI**gher or **S**ame — переход, если выше или столько же).

Допустим, нам надо обнулить блок памяти, на начало которого указывает **R0**, а на конец — **R1**. До изучения этого параграфа мы, вероятно, попытались бы сделать это так:

```

      LOOP:      CLR      (R0)+
                CMP      R1,R0
                BPL      LOOP

```

Рассмотрим, однако, что произойдет, если **R0** указывает на ячейку **60000**, а **R1** — на ячейку **170000**, предполагая, конечно, что в нашем распоряжении имеется достаточное количество памяти, чтобы эти адреса существовали. Первая команда **CMR** будет выполнена с **R0**, содержащим **60002**, и в результате операции получится число **107776**. Его пятнадцатый разряд равен единице, следовательно, будет установлен бит *N*, и команда **BPL** не приведет к переходу. Еще раз предостерегаем от неявного допущения, что процессор «знает», с каким типом данных он работает (как в данном случае — что он работает с адресами, а не с обычными числами со знаком).

Нужно повторять цикл до тех пор, пока при выполнении команды **CMR** нулевой регистр не станет содержать **170002**. На последнем этапе операция сравнения будет выглядеть так:

$$\begin{array}{r}
 170000 \\
 - 170002 \\
 \hline
 177776
 \end{array}$$

Для ее завершения требуется перенос единицы в левый разряд двоичного представления числа **170000** и, как следствие, установка бита *C*. До тех пор, пока значение **R0**, рассматриваемое как шестнадцатиразрядное положительное число, не превосходит **170000**, для выполнения операции вычитания в подобном переносе нет необходимости, и потому бит *C* всегда будет равен **0**. Итак, наша программа должна быть такой:

```

      LOOP:      CLR      (R0)+
                CMP      R1,R0
                BCC      LOOP

```


В данном случае мнемоника **BHIS** предпочтительнее, чем **BCC**.

По команде **BLOS** — переход, если ниже или столько же (Branch if LOwer or Same) — переход происходит, если разряды **C** или **Z** (или оба) установлены: $C \vee Z = 1$. Дополнительной к ней является команда **BHI** (Branch if HIgher — переход, если выше).

Установка условных признаков. Программа может по своему усмотрению устанавливать или сбрасывать четыре бита-признака. Левые одиннадцать битов кода операции соответствующих команд содержат **00024** (почему одиннадцать?). Биты 0, 1, 2 и 3 устанавливаются в зависимости от того, к какому из признаков **C**, **V**, **Z** или **N** относится команда. Если четвертый бит кода команды равен нулю, то выбранные признаки сбрасываются, в противном случае — устанавливаются. Для сброса или установки отдельных признаков имеются команды со специальной мнемоникой: **CLC**, **CLV**, **CLZ**, **CLN** и **SEC**, **SEV**, **SEZ**, **SEN**. Одновременно все четыре признака сбрасываются командой **CCC**, а устанавливаются командой **SCC**. Код **000240** соответствует команде **NOP** (нет операции) — она ничего не изменяет.

Во время трансляции ассемблер может выполнять логическую операцию *и* (обозначается символом **&**) и операцию *включающего или* (обозначение — **!**). Так, в результате трансляции **CLV!CLC** будет получено слово, в котором равны 1 биты, соответствующие установленным битам в командах **CLV** или **CLC** (или в обоих одновременно). Следовательно, такая комбинация команд осуществляет сброс битов **V** и **C**.

- УПРАЖНЕНИЯ. 1. Как выглядит код команды а) **CLV**? б) **CLC**?
 2. Какие действия будет выполнять команда **CLV&CLC**?
 3. Что будет делать команда **SEV!CLC**?

Байтовые команды. Результат трансляции команд **MOV**, **CLR**, **INC**, **DEC**, **NEG**, **BIT**, **BIC**, **BIS**, **COM**, **TST** и **CMR** содержит нулевой пятнадцатый бит. Если к мнемонике любой из этих команд добавляется буква **B** (**MOVB**, **CLRB** и т. д.), то пятнадцатый бит кода операции команды равен 1, и команда становится байтовой. С небольшими предосторожностями мы можем применять их точно так же, как и соответствующие команды, работающие со словами.

Отрицательные числа представляются в байте в восьмиразрядном дополнительном коде. Так, если первоначально в слове с меткой **MEM** был записан нуль, то после выполнения команды

MOVB # - 2, MEM

оно будет содержать 000376. В то же время команда (если опять-таки первоначально ячейка **МЕМ** была нулевой)

MOVB #-2,МЕМ+1

оставит младший байт без изменения, а все разряды старшего байта, за исключением самого правого, станут равны 1. Следовательно, ячейка **МЕМ** будет содержать 177000. (Замечание. Но не 376000 — середина слова не совпадает с границей восьмеричной цифры!)

УПРАЖНЕНИЕ. Что будет содержать ячейка **МЕМ** после выполнения команды **MOVB #-1,МЕМ+1**, если первоначально в ней были нули?

Условные признаки устанавливаются в соответствии с результатом байтовой операции. В частности, бит **N** устанавливается в зависимости от состояния старшего разряда того байта, с которым производилась операция (седьмой разряд, если работа шла с младшим байтом).

Команда **СМРВ** установит бит **С** в зависимости от того, был ли перенос **1** при выполнении восьмиразрядной операции вычитания в самый левый разряд. Заметьте, что команды **ADD** и **SUB** не имеют байтовых аналогов.

Как правило, байтовая команда не затрагивает содержимое другого байта того слова, половина которого является ее приемником. Единственным исключением является команда **MOVB** с регистром в качестве приемника. (Не забывайте, что байтовая команда может адресоваться только к младшему байту регистра.) Она пересылает данные в младший байт регистра и одновременно устанавливает каждый разряд старшего байта равным значению седьмого разряда. То есть при занесении в регистр команда **MOVB** распространяет знаковый разряд младшего байта на все слово. Если пересылаемые данные рассматривались как целые числа со знаком, то содержимое регистра будет правильным при обращениях к нему и как к байту, и как к слову.

Если в байтовых командах применяется автоинкрементная или автодекрементная адресация, то, как правило, ЦП изменяет значение регистра на 1, а не на 2. Существуют, однако, важные исключения, при которых содержимое регистра изменяется на 2 даже в байтовых командах, а именно: когда регистр есть **SP** или **PC** и когда используется косвенная адресация.

УПРАЖНЕНИЕ. Как вы думаете, какими соображениями руководствовались разработчики машины **PDP-11**, предусматривая эти исключения в аппаратуре?

Учтите, что для работы таких команд, как **JSR** и **RTS**, указатель системного стека **SP** всегда должен указывать на слово. Поэтому для заведения в программе байтового стека под его указатель необходимо отвести отдельный регистр.

Команда перестановки байтов **SWAB** (SWAp Bytes) меняет местами два байта слова-приемника. Если ячейка **MEM** содержит код **000376**, то после выполнения команды

SWAB MEM

в ней будет **177000**. Команда **SWAB** сбрасывает биты **V** и **C**, а биты **N** и **Z** устанавливает в соответствии с *исходным* значением старшего байта приемника. Иными словами, **SWAB** устанавливает, например, бит **N** соответственно значению седьмого разряда результата.

Команды арифметического и циклического сдвигов. Довольно часто необходим доступ к информации, содержащейся в самой комбинации битов слова. Допустим, нам нужно знать *поразрядную четность* слова, т. е. выяснить, четное или нечетное количество битов, равных **1**, в нем содержится. Это необходимо, например, для контроля правильности содержимого слова. С этой целью в каждом слове файла резервируется один бит (контрольный бит), значение которого (**0** или **1**) выбирается так, чтобы в слове оказалось, скажем, нечетное количество единиц. Если в дальнейшем файл будет случайно испорчен, то существует вероятность (какая?) того, что в каком-то слове четность нарушится, и последующий контроль позволит это выявить.

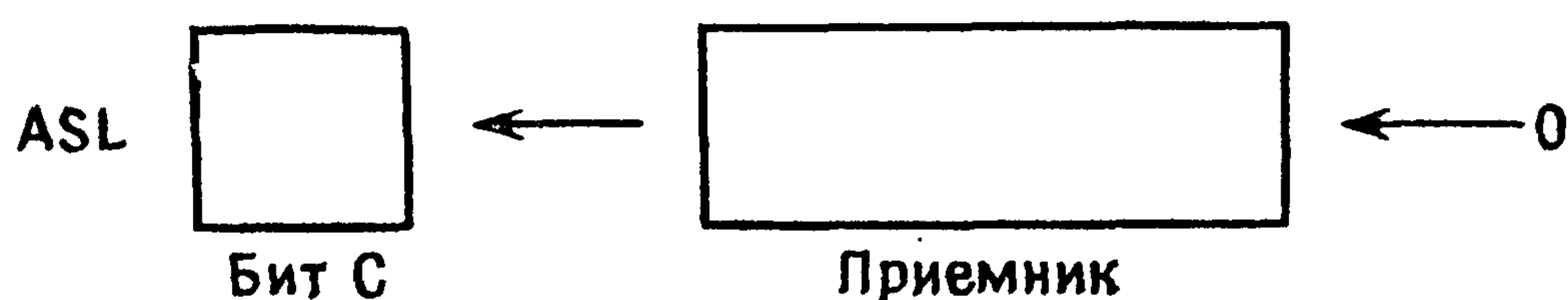
Предположим, что требуется проверить на четность содержимое регистра **R1**, после чего записать в **R0** единицу, если число единичных битов в **R1** нечетно, и нуль в противном случае. Вначале очистим **R0**, а затем будем по очереди проверять каждый бит **R1** и при обнаружении очередной единицы менять значение **R0**: на **0**, если в нем была **1**, и наоборот. Для выполнения этой операции над **R0** мы сначала запишем в **R2** единицу, а затем (при обнаружении единицы в **R1**) будем выполнять команду

XOR R2,R0

Теперь нужно научиться проверять каждый бит **R1**. Существуют четыре команды, приспособленные для последовательного анализа битов. Все они самым непосредственным образом используют бит **C**.

Команда арифметического сдвига влево **ASL** (Arithmetic Shift Left) выполняет над содержимым слова-приемника следующую операцию: значение разряда **15** засылается в бит **C**; значение раз-

ряда 14 — в разряд 15; значение разряда 13 — в разряд 14 и т. д. до нулевого разряда, значение которого пересылается в первый разряд. Нулевой разряд сбрасывается. На диаграмме это можно изобразить так:



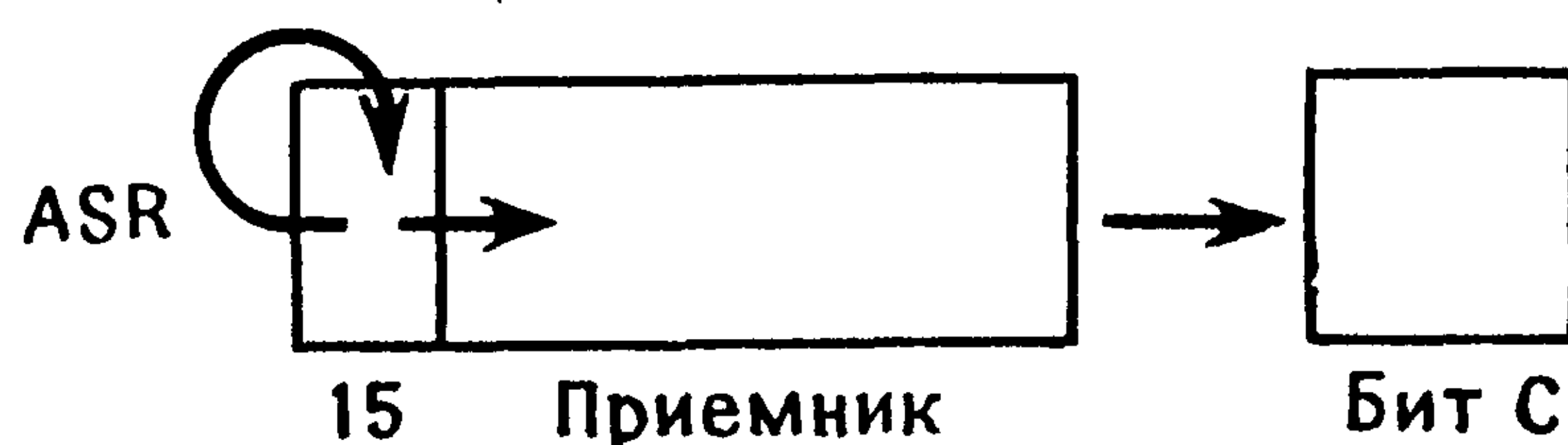
Команда **ASL** позволяет эффективно выполнять операцию умножения на два. Признаки **N** и **Z** устанавливаются или сбрасываются в зависимости от значения результата. Бит **V** устанавливается, если только один из признаков (**N** или **C**) установлен; в противном случае бит **V** сбрасывается. Таким образом, $V = N \vee C$.

УПРАЖНЕНИЯ. 1. Правильно ли команда **ASL** умножает на два отрицательные числа, представленные в дополнительном коде?

2. Согласуется ли состояние бита **V** с арифметическим результатом операции?

Мнемоника байтовой версии команды **ASL** есть **ASLB**. Она сдвигает старший разряд байта в бит **C** и сбрасывает его младший разряд. Остальные условные признаки устанавливаются так же, как и командой **ASL**, но в зависимости от результата байтовой операции.

Команда арифметического сдвига вправо **ASR** (Arithmetic Shift Right) заносит в бит **C** значение нулевого разряда приемника, а значение каждого следующего разряда сдвигает на одну позицию вправо. Значение разряда 15 засылается в разряд 14, а *само при этом не изменяется*.



Байтовая команда **ASRB** работает аналогичным образом. Она не изменяет состояние старшего разряда байта. Условные признаки устанавливаются обеими командами так же, как и соответствующими им командами сдвигов влево.

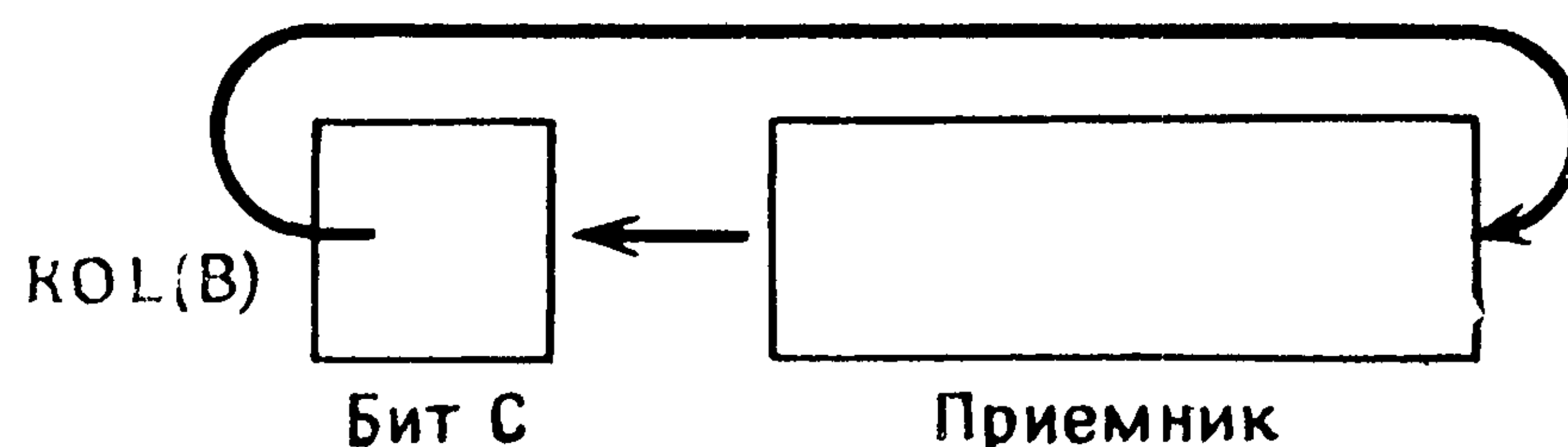
УПРАЖНЕНИЯ. 1. Для выполнения какой арифметической операции предназначена команда **ASR**?

2. Для любых ли чисел она выполняется правильно?

3. Характеризует ли состояние бита **V** корректность результата?

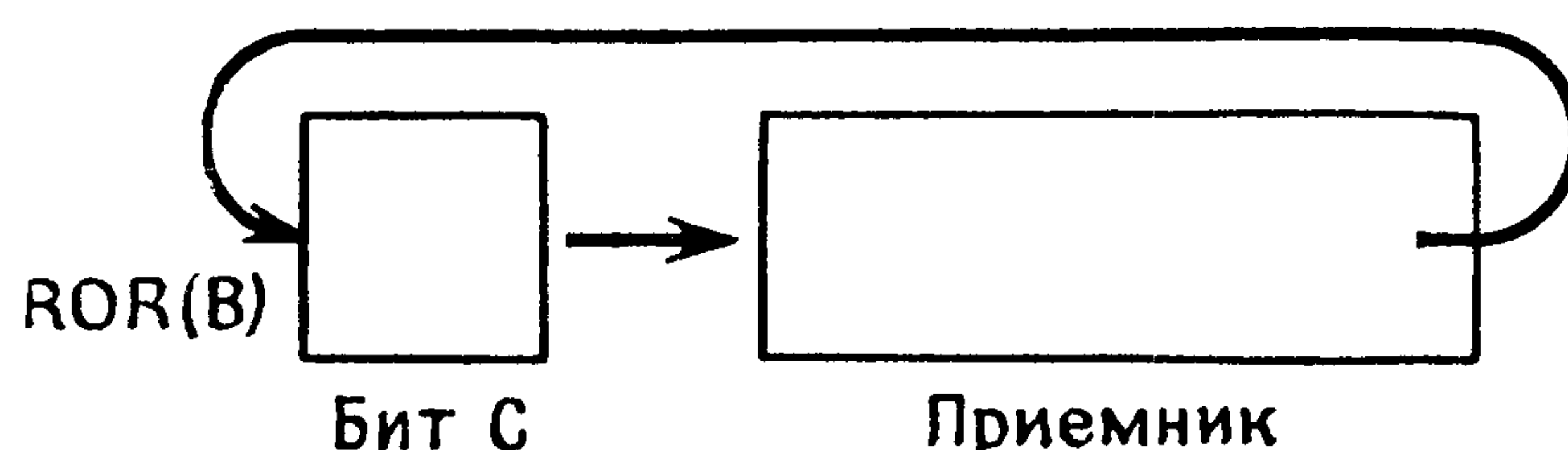
4. Можете ли вы теперь написать программу проверки на четность?

Команды циклического сдвига влево слова **ROL** (ROtate Left) и байта **ROLB** производят следующую операцию:



Значение каждого разряда сдвигается на одну позицию влево; значение старшего разряда переходит в бит С, а значение бита С — в младший разряд.

Аналогично команды циклического сдвига вправо слова **ROR** (ROtate Right) и байта **RORB** выполняют операцию



Значение каждого разряда сдвигается на одну позицию вправо; состояние младшего разряда переходит в бит С, а значение бита С записывается в старший разряд.

Условные признаки N, Z и V устанавливаются этими командами точно так же, как и командами арифметических сдвигов.

УПРАЖНЕНИЕ. Можете ли вы догадаться, почему команды циклических сдвигов реализованы так, что они устанавливают $V = N \vee C$?

Чтобы проиллюстрировать применение команд циклических сдвигов, допустим, нам известно, что по адресу **X** расположена двухадресная команда, а нам нужно, не меняя содержимого ячейки **X**, занести в нулевой регистр номер режима адресации к источнику в этой команде. Из § 2.4 мы знаем, что режим адресации к источнику дается в разрядах слова с 9 по 11. Поэтому одно из решений заключается в том, чтобы после засылки **MOV X, R0** девять раз повторить команду **ROR R0** и сдвинуть таким образом разряды с 9 по 11 на место разрядов с 0 по 2. Теперь команда **BIC #177770, R0** сотрет все остальные разряды в **R0**. Однако лучше сделать так:

MOVB	X+1, R0
ROR	R0
BIC	#177770, R0

Прекрасное обсуждение общей проблемы перемещения данных из части одного слова в часть другого применительно к машине PDP-11 можно найти в книге: Wulf et al., *The Design of an Optimizing Compiler* (American Elsevier, 1975). Как отмечают ее авторы, универсального подхода не существует, и, чтобы в нашем случае найти наиболее эффективное решение, требуется определенная изобретательность. Попробуйте разобраться в приведенном ниже способе пересылки содержимого девяти правых разрядов ячейки **Y** в девять левых разрядов ячейки **X** без изменения содержимого **Y** и остальной части ячейки **X**:

MOV	Y,R0
ROLB	X
ROR	R0
RORB	X
MOVB	R0,X+1

Этот фрагмент, который занимает всего девять слов, принадлежит автору настоящей книги. Если у вас есть желание, то попробуйте его улучшить.

УПРАЖНЕНИЯ. 1. Составьте программу, которая устанавливает а) бит **C**, б) бит **N**, если в двухадресной команде по адресу **X** используется режим косвенной адресации к источнику, и сбрасывает этот бит в противном случае. Содержимое **X** не изменяйте.

2. Напишите программу, которая заносит в регистр **R0** число, равное количеству слов, занимаемых двухадресной командой, начинающейся в ячейке **X**.

Коды команд перехода. Все команды перехода транслируются в одно машинное слово. Старший байт содержит код операции, а содержимое младшего байта определяет адрес перехода. Последний задается как смещение относительно текущего значения счетчика команд **PC**. Так, предположим, что в ячейке **2000** находится команда **BNE LABEL**, где метка **LABEL** относится к ячейке **2010**. Когда выполняется команда перехода, **PC** содержит адрес следующего слова, т. е. **2002**. Смещение от команды до места перехода составляет $2010 - 2002 = 6$ байтов (все числа восьмеричные!) или три слова. Ассемблер закодирует смещение, выраженное в словах, в младший байт команды. (Конечно, во время исполнения команды этот факт учитывается.) Код операции команды **BNE** содержит **1** в девятом разряде и **нуль** в остальных разрядах старшего байта. Следовательно, в нашем случае команде **BNE LABEL** соответствует код **001003**.

Величина смещения, хранящаяся в младшем байте команды ветвления, представляется в форме положительного или отрица-

тельного числа, причем в последнем случае как восьмиразрядное число в дополнительном коде. Допустим, мы хотим поместить в ячейку 2010 команду **BNE LOOP**, где **LOOP** имеет адрес 2000. В момент исполнения команды перехода счетчик команд содержит адрес 2012. Следовательно, смещение от команды до места перехода равно $2000 - 2012 = -12$ байтов или -5 (восьмеричных!) слов. Отсюда видно, что смещение назад всегда оказывается отрицательным. Код числа -5 (в младшем байте) равен 373, и потому код команды **BNE LOOP** имеет вид 001373. Встретив такой код в листинге программы, мы узнали бы в нем команду перехода *назад на пять слов, отсчитывая от слова, следующего за командой перехода*.

Легко видеть, что значение младшего байта, равное 177, соответствует переходу на 200 (D 128) слов вперед по отношению к самой команде, а его значение, равное 200, представляет собой число -200 и соответствует максимальному переходу на 177 (D 127) слов назад от команды. Одной командой перехода нельзя передавать управление за пределы этого диапазона.

УПРАЖНЕНИЯ. 1. Команда **BEQ** в восьми левых разрядах содержит код операции 0014. Каков был бы результат трансляции команды **BEQ LABEL**, расположенной в ячейке 2006, если бы метка **LABEL** относилась к ячейке а) 2000, б) 1712, в) 2076?

2. Являются ли команды перехода позиционно-независимыми?

Остается еще одна, последняя команда перехода **SOB**¹⁾ (Subtract One and Branch if nonzero — вычитание единицы и переход, если не нуль). Ее код операции 077 (с 9 по 15 разряды), номер регистра указывается в разрядах с 6 по 8, а смещение — в разрядах с 0 по 5. Это смещение вычисляется как шестизначное положительное число, но интерпретируется как смещение *назад*. Таким образом, командой **SOB** нельзя передать управление вперед. Она вычитает единицу из содержимого указанного регистра и производит передачу управления в том случае, если результат не равен нулю. Команда **SOB** полезна для организации циклов. Например, мы можем очистить память, начиная с ячейки **MEM** и до ячейки **WRD** включительно, следующей последовательностью команд:

```

                                MOV      #MEM,R0
                                MOV      #<WRD - MEM>/2 + 1,R1
LOOP:                          CLR      (R0)+
                                SOB      R1,LOOP

```

УПРАЖНЕНИЯ. 1. На какое максимальное расстояние можно передать управление в команде **SOB**?

¹⁾ Отсутствует на некоторых малых моделях,

2. Как кодируется команда **SOB** в приведенном выше примере?
3. Напишите программу очистки:
 - а) тысячи ячеек, кончая ячейкой **MEM** (включительно);
 - б) ячеек, начиная с **MEM — 1000** и до **MEM** включительно.

3.4. Модульное программирование

В этом параграфе дальнейшее развитие получит тема § 3.1 и 3.2, касающаяся разбиения программируемой задачи на более обозримые и простые части, или *модули*. Прежде всего нас будет интересовать связь между модулями.

Когда в § 3.2 мы составляли программу вычисления арифметических выражений, то договорились, что выражения должны считываться в блок по указателю **R2**, а вычисления — выполняться в стеке, на который указывает **R1**. К примеру, программа **NUM** была устроена так, что извлекала данные из ячеек, начиная с (**R2**), а результат заносила в **—(R1)**. Такое построение обеспечивает более широкое ее применение для чтения чисел, а не только в узких рамках конкретной задачи. Подпрограмма **NUM** может брать цифры из любого места памяти, а полученное число заносить в любую ячейку. Все, что нам нужно сделать, — это перед вызовом подпрограммы **NUM** установить указатели **R1** и **R2**.

Передача параметров. Те элементы информации, которые должны быть доступны подпрограмме, чтобы она могла выполнить свою функцию, называются *параметрами*. Вызывающая процедура должна *передать* параметры подпрограмме. Согласно данному определению, адрес возврата, очевидно, является параметром, и мы уже обсудили, как он передается в подпрограмму командой **JSR**. Существует несколько способов передачи параметров, которые позволяют внутри подпрограммы определить, откуда брать данные и куда пересылать результаты. При программировании на языке ассемблера применение любого из них не вызовет никаких затруднений, поскольку в этом случае программист точно знает, где что происходит. Гораздо легче запутаться при программировании на языках высокого уровня, так как здесь механизм от нас скрыт и неизвестно, какой в действительности способ применяется.

Допустим, мы пишем подпрограмму **SQRT** для вычисления квадратного корня из числа. Рассмотрим, каким образом можно передавать ей параметры. Подпрограмма может извлекать квадратный корень из числа, находящегося в ячейке, на которую указывает **R1**, а результат заносить в ячейку, на которую указывает **R2**. Это похоже на способ, использованный в подпрограмме **NUM**. Для записи в ячейку **WRD** квадратного корня из числа,

хранящегося по адресу **MEM**, мы должны написать

```
MOV      #MEM,R1
MOV      #WRD,R2
JSR      PC,SQRT
```

Чтобы не затереть исходное число, подпрограмма **SQRT** должна начинаться примерно так:

```
MOV      (R1),R1
```

Если квадратный корень вычислялся в **R1**, то подпрограмму можно заканчивать командами

```
MOV      R1,(R2)
RTS      PC
```

Ясно, что при желании ее нетрудно изменить так, чтобы квадратный корень из числа, находящегося в ячейке **MEM**, заносился туда же.

Другой способ передачи параметров состоит в их перечислении вслед за командой **JSR**, но в этом случае регистром связи не должен быть **PC**:

```
JSR      R5,SQRT
.WORD    MEM,WRD
```

Подпрограмма считывает данные в **R1**

```
MOV      @(R5)+,R1
```

а результат пересылает из **R1**

```
MOV      R1,@(R5)+
```

Автоинкрементный режим гарантирует нам, что параметры будут передаваться последовательно; он также позволяет осуществить возврат командой **RTS R5** на команду, стоящую сразу за последним параметром. Однако если параметры передаются по-другому, то редко приходится выделять особый регистр для связи подпрограмм.

Можно само число, из которого извлекается корень, поместить сразу после команды **JSR**, а результат записать в ячейку **WRD**:

```
JSR      R5,SQRT
.WORD    7                ; вычислить корень из 7
.WORD    WRD              ; и занести его в WRD
```

если подпрограмма прочтет свой параметр командой

```
MOV      (R5)+,R1
```


Можно также перед вызовом подпрограммы заносить параметры в системный стек. В этом случае необходимо проявить очевидную осторожность и обеспечить, чтобы при выходе **SP** указывал на адрес возврата, а не на параметр. Нужно также не оставлять параметры в стеке, иначе они быстро его заполнят. Для этого может понадобиться подпрограмма, которая изменяя регистр **SP** в соответствии с количеством передаваемых параметров, «очищает» стек. Во избежание нежелательных последствий следует соблюдать аккуратность при подсчетах.

УПРАЖНЕНИЯ. 1. Какие соображения могли бы заставить вас предпочесть один из рассмотренных способов передачи параметров другим? В частности, который из них предпочтительнее для: а) вложенных подпрограмм; б) рекурсивных подпрограмм; в) подпрограмм с большим числом параметров; г) подпрограмм с переменным количеством параметров?

2. Доведите до совершенства ваши подпрограммы: а) умножения двух чисел; б) деления одного числа на другое; в) чтения десятичного числа с терминала; г) вывода десятичного числа на терминал.

3. Как должны передавать друг другу параметры подпрограммы? Есть ли необходимость после них «подчищать» стек?

Параметры в языках высокого уровня. Тот, кто программирует на языке ассемблера, заинтересован в выборе наиболее подходящего и эффективного метода связи подпрограмм и должен уделять внимание техническим подробностям. С другой стороны, тех, кто программирует на языках высокого уровня, совсем не заботят какие бы то ни было детали связи подпрограмм, но зато им постоянно грозит опасность логической ошибки, если они не представляют себе, какой способ взаимосвязи подпрограмм реализован в используемом ими языке.

Программирующим на языках высокого уровня мало дела до того, какой из только что рассмотренных методов передачи параметров применяется. Их интересует не то, как подпрограмма получает информацию, а что это за информация и как с ней работать.

При программировании на языке высокого уровня можно рассуждать исключительно в арифметических терминах, в терминах переменных и констант. Языки с такими синтаксическими конструкциями, как $X=3$, создают впечатление, что некий абстрактный объект X мгновенно приравнивается значению 3. Языки с синтаксисом вида $X \leftarrow 3$ или $X:=3$ более явно отражают реальность: вместо фразы «положить X равным 3» мы читаем приведенную строчку так: «занести 3 в X ». Когда программист объявляет X переменной, транслятор отводит под нее ячейку

памяти, которой дает имя **X**. Ссылки на **X** в языке высокого уровня транслируются затем в машинные команды, затрагивающие содержимое ячейки **X**. Так, результат трансляции $Y = X + 1$ мог бы быть таким:

```
MOV      X,Y
INC      Y
```

в то время как кажущаяся парадоксальной запись $X = X + 1$ есть просто-напросто языковой вариант команды **INC X**.

Очевидный способ для транслятора машины PDP-11 перевести оператор $X = 3$ в **MOV #3,X**, и это действительно лучшее решение. Рассмотрим, однако, и другую возможность:

```
MOV      THREE,X
...
...
THREE:   .WORD      3
```

Здесь расходуются четыре слова вместо трех, и потому ясно, что это решение хуже. Мы упомянули о нем только потому, что на других вычислительных машинах подобная методика обладает преимуществами. Не на всех вычислительных машинах предусмотрен непосредственный режим адресации. Ассемблер тех машин, на которых он отсутствует, обрабатывая конструкцию, эквивалентную команде **MOV #3,X**, отведет в специальном блоке памяти слово для хранения константы 3 (ассемблер заведет *литерал*), а при трансляции воспользуется вторым из приведенных выше способов. Хотя разговор о резервировании памяти для констант не имеет отношения к машине PDP-11, он представляет здесь интерес постольку, поскольку во многих работах считается, что написанная на языке высокого уровня инструкция $Y = X + 3$ будет всегда транслироваться чем-то вроде

```
MOV      X,Y
ADD      THREE,Y
```

где **THREE** есть ячейка, содержащая число 3, которое, таким образом, оказывается доступным всякий раз, когда в программе встречается константа 3. Это действительно справедливо по отношению к хорошим трансляторам на многих машинах, поскольку может оказаться, как было показано выше, более эффективным транслировать программу именно таким способом. Рассмотрим теперь, что произойдет, если во время трансляции встретится инструкция $3 = 3 + 3$ и транслятор не заметит, что она ошибочная. Тогда сгенерируются такие команды:

```
MOV      THREE,THREE
ADD      THREE,THREE
```


хотя в хорошем трансляторе на стадии оптимизации избыточность первой строки будет опознана. При исполнении программы любая последующая ссылка на константу 3 приведет к выборке числа 6. Испортить подобным образом константу в некоторых языках гораздо проще, чем это может показаться из нашего прозрачного примера.

На машине PDP-11, однако, инструкция $3=3+3$ была бы (если бы позволял язык) оттранслирована в команду **MOV #3, #3**, которая безобидна. Как она будет исполняться?

Допустим теперь, что мы располагаем функцией извлечения квадратного корня и что нам захотелось на языке высокого уровня иметь возможность написать

$$Y = \text{SQRT}(X)$$

не изменяя при этом значение **X**. Последнее требование означает, что подпрограмме запрещено работать непосредственно с содержимым ячейки **X**, поэтому она *скопирует* его в свою рабочую область, скажем в регистр

MOV X,R0

и уже не будет более ссылаться на **X**. Говорят, что параметр **X** *вызывается по значению*.

Учтите, что **Y** не является параметром. Подпрограмма **SQRT** пишется так, что результат остается именно в той ячейке, которую транслятор использует, если за оператором **=** следует имя функции. Так, если нужно написать **SQRT** и известно, что «**Y=**», предшествующее имени функции, транслируется в **MOV R0, Y**, то значение квадратного корня из **X** следует занести в регистр **R0**.

Функцию извлечения корня можно переписать так, что она будет вызываться как подпрограмма или процедура. В этом случае ее вызов выглядит, например, так:

CALL SQRT(X,Y)

Подпрограмма извлекает квадратный корень из содержимого ячейки **X** и заносит результат в ячейку **Y**. Здесь уже **Y** является параметром подпрограммы, при этом говорят, что он *вызывается по результату*. Заметьте что параметр может вызываться одновременно и по значению, и по результату.

Классическим примером подпрограммы, параметры которой нельзя передавать по значению, является процедура взаимного обмена между содержимым двух переменных:

CALL EXCH(X,Y)

Подпрограмма, которая копирует содержимое X и Y в свою рабочую область, меняет местами содержимое этих двух рабочих ячеек и затем осуществляет возврат, особого интереса для нас не представляет. Разумеется, подпрограмма должна работать *непосредственно* с указанными в вызове ячейками. Мы говорим, что X и Y *вызываются по ссылке*.

Можно придумать патологические примеры подпрограмм, параметры которых вызываются или по значению-результату, или по ссылке — с непредвиденными в обоих случаях последствиями. Попробуем перевести на язык ассемблера подпрограмму **DIFSUM(X, Y)**, которая на языке высокого уровня имеет вид

$$\begin{aligned} X &= X - Y \\ Y &= 2 * Y + X \end{aligned}$$

и которая должна занести разность значений X и Y в X , а их сумму в Y . Обратите внимание, что во второй строке подразумевается, что величина X имеет уже новое значение, полученное в первой строке. Мы могли бы вызвать X и Y по значению-результату

MOV	X,R0	; вызов по
MOV	Y,R1	; значению
SUB	R1,R0	
ADD	R1,R1	
ADD	R0,R1	
MOV	R0,X	; вызов по
MOV	R1,Y	; результату
RTS	PC	

или по ссылке

MOV	#X,R0	; вызов по
MOV	#Y,R1	; ссылке
SUB	(R1),(R0)	
ADD	(R1),(R1)	
ADD	(R0),(R1)	
RTS	PC	

Теперь проследите, что произойдет в обоих случаях при вызове **DIFSUM(X, X)**.

УПРАЖНЕНИЯ. 1. Если подпрограмма имеет только один параметр, есть ли разница между тем, как он вызывается: а) по значению-результату или б) по ссылке?

2. Изучите способы передачи параметров, применяемые в наиболее знакомом вам языке высокого уровня.

Программные прерывания. Основное назначение программно-го прерывания состоит в автоматическом вмешательстве ЦП в

ход выполнения программы, когда возникают потенциально опасные ситуации. В PDP-11 эта трактовка расширена и включает дополнительные возможности для связи программных модулей, прежде всего в интересах пользователей изолированных систем.

Попробуйте вставить в программу несуществующую команду; этому прекрасно послужит, к примеру, «команда» **.WORD 7**, поскольку такой код не соответствует никакой команде. Программа остановится, причем, если она исполнялась под управлением монитора, будет напечатано сообщение о том, что произошло *прерывание в ячейке 10*.

Когда встречается неизвестная команда, аппаратура автоматически выполняет действия, эквивалентные командам

```
MOV      PS, -(SP)
MOV      PC, -(SP)
```

т. е. пересылает слово состояния процессора **PS** и счетчик команд **PC** в системный стек, а затем в них заносит новое содержимое, выполняя действия, соответствующие таким командам:

```
MOV      10, PC
```

```
MOV      12, PS
```

Значения адресов **10** и **12** встроены в аппаратуру, и программист их изменить не может. Говорят, что по несуществующей команде происходит программное прерывание в ячейке **10**, где расположен *вектор прерывания, состоящий из двух слов*.

Во время загрузки операционной системы десятая ячейка настраивается так, чтобы она указывала на программу действий в подобной ситуации. Эта программа называется *программой обслуживания (обработки) прерывания*. В нашем случае в ней просто выдается на терминал соответствующее сообщение и осуществляется выход на уровень команд монитора.

Далее, как мы уже упоминали, на некоторых моделях машины PDP-11 команды **MUL** и **DIV** отсутствуют. Поэтому в программе, которая исполняется на таком процессоре и в которой есть эти команды, возникнет прерывание в ячейке **10**. Заметьте, что запуск такой программы вообще невозможен, если ассемблер не умеет распознавать команды **MUL** и **DIV**; тогда он просто не сможет их закодировать. Можно, конечно, вставить в программу собственно коды этих команд с помощью директивы **.WORD**.

Как видно, у пользователя, который разрабатывает программу и предполагает использовать ее на различных моделях PDP-11, есть проблемы. Естественно, ему не хочется переписывать программу при переносе ее на другую машину. Более того, неразумно тратить машинные ресурсы даже на перетрансляцию программы. Она должна храниться на ленте, переносном диске или дру-

гом запоминающем устройстве в виде образа памяти и быть готовой к немедленной загрузке и запуску. Одно решение заключается в том, чтобы совсем отказаться от команд **MUL** и **DIV**. Однако эти команды намного быстрее, чем соответствующие им программные реализации; следовательно, в этом случае опять-таки неизбежны потери.

Можно поступить более гибко, а именно свободно использовать в программе команды **MUL** и **DIV**, но предварительно подготовить десятую ячейку так, чтобы для выполнения этих операций управление передавалось на соответствующий фрагмент в самой программе. Таким образом, если в машине предусмотрены команды **MUL** и **DIV**, то они будут выполняться непосредственно, если же нет, то через десятую ячейку управление будет передаваться программам пользователя.

В изолированной системе или для привилегированной программы в системе с разделением времени десятая ячейка доступна, как и любая другая ячейка памяти. Следовательно, адрес начала программы обслуживания прерываний **SERV** можно занести в эту ячейку командой **MOV #SERV, 10**. (Как еще можно это сделать?) Учтите, что если монитор отсутствует, то программа обязательно должна подготовить системный стек (почему?)

Программа обслуживания (программного) прерывания. Программа **SERV** должна (зная адрес возврата) извлечь команду, вызвавшую прерывание. Затем она должна определить, какая это команда: **MUL** или **DIV**. Команда **MUL** имеет код **070** в семи левых битах, за которыми следуют три бита под регистр-приемник и шесть битов под источник. Аналогично кодируется и команда **DIV**, только ее код операции равен **071**. Нужно также позаботиться о том, чтобы в случае какой-либо иной причины прерывания программа **SERV** выдавала подходящее сообщение об ошибке. Если же причиной прерывания является команда **MUL** или **DIV**, управление должно быть передано на программу соответственно умножения или деления, после завершения которых все в машине должно выглядеть в точности так же, как если бы команды выполняла аппаратура.

Программам умножения и деления должны быть переданы значения операндов. Для примера допустим, что в программе стоит команда

X: **MUL** Y,R

где **R** — один из регистров, а **Y** — выражение, которое обозначает адрес и требует, возможно, индексной или косвенной адресации. Конечно, адрес команды (с меткой **X**) программа обслужи-

вания прерываний должна взять из стека, но для большей простоты изложения будем полагать, что он уже известен.

Регистр **R** может быть любым из первых шести регистров, так как нельзя написать команд, в которых употребляются неизвестные регистры (*так ли уж это невозможно?*), поэтому мы должны начать, скажем, с команды **MOV R, R0**. Это можно сделать, переписав разряды, задающие регистр **R** в команде **MUL**, в поле источника команды **MOV** до выполнения последней. К счастью, для этого требуется просто перенести разряды 6—8 команды **MUL** в те же самые разряды команды **MOV**:

```

                                MOV      X,R0
                                BIC      #177077,R0
                                ADD      R0,L1
L1:                            MOV      R0,R0

```

(Будет ли это работать, если окажется, что **R** совпадает с **R0**?) Не противоречит ли такое использование самомодифицирующихся программ утверждению, приведенному в § 3.3, о том, что транслятор должен оптимизировать программу, исключая из нее команды **MOV**, в которых источник и приемник совпадают?

Второй операнд добыть не столь легко. В качестве первого шага мы можем занести биты поля приемника (с 0 по 5) из ячейки **X** в биты источника (с 6 по 11) команды «**MOV** в **R1**»:

```

                                MOV      X,R1
                                BIC      #177700,R1
                                SWAB     R1
                                ASR      R1
                                ASR      R1
                                ADD      R1,L2
L2:                            MOV      R0,R1

```

Заметьте, что, как и ранее, первоначальный источник команды с меткой **L2** выбран таким просто для того, чтобы соответствующее ему поле было нулевым и оказалось подготовленным к немедленной модификации предшествующей командой **ADD**.

Все хорошо, если команда **MUL** укладывается в одно слово, да еще и операнд источника не совпадает с **R1**. (Что нужно изменить, чтобы учесть случай, когда операнд источника есть **R1**?) Допустим, однако, что режим адресации источника приводит к трансляции команды **MUL** в два слова. Точно такой же режим адресации появится теперь в команде **MOV** (с меткой **L2**), и, следовательно, при вычислении адреса источника будет использоваться слово, *следующее* за **L2**. Чтобы учесть это, мы можем перенести второе слово команды **MUL** и заменить им последнюю.

строку следующего фрагмента:

	MOV	X+2,L2+2
L2:	MOV	R0,R1
	HALT	

Конечно, останов здесь «фиктивен»: команда по адресу **L2—6** (где находится этот адрес?) гарантирует, что исполняться он не будет.

УПРАЖНЕНИЕ. Трудность, связанная с этой последней попыткой, заключается в том, что если теперь **MUL** транслируется в одно слово, то, какая бы команда ни оказалась в **X+2**, она будет пересылаться в ячейку **L2+2**. Измените программу так, чтобы команда **MOV X+2,L2+2** выполнялась в ней, если только **MUL** состоит из двух слов. (Можете заменить **HALT** на **NOP**, если вам это больше нравится.)

Даже теперь наша программа не сможет справиться с задачей, если в команде **MUL** применяется относительная (режим 6 с использованием **PC**) или косвенно-относительная (режим 7 с использованием **PC**) адресация, потому что в этом случае ячейка **X+2** будет содержать адрес источника или указатель на него *относительно адреса расположенной в X команды*.

УПРАЖНЕНИЕ *. Измените соответствующим образом вашу программу. Это послужит прекрасным упражнением для повторения различных способов адресации.

Программа обслуживания прерываний должна заканчиваться командой выхода из прерывания **RTI** (ReTurn from Interrupt), которая реализует в виде одной операции действия, эквивалентные последовательности двух команд:

MOV	(SP)+,PC
MOV	(SP)+,PS

Предупреждение. Программа обслуживания прерываний, написанная пользователем, может применяться и в присутствии монитора, но тогда необходимо сохранять ячейки 10 и 12, а перед выходом восстанавливать их, чтобы не потерять путь в мониторную программу обработки.

УПРАЖНЕНИЕ *. Завершите программу обслуживания прерывания, моделирующую аппаратные команды умножения и деления.

Абсолютная адресация. Можно обойтись и без относительной адресации, если воспользоваться режимом 3 и задать **РС** в качестве регистра. Тогда говорят об *абсолютной адресации*, а в команде ее обозначают двумя символами **@#**, предшествующими адресу. Так, команда

CLR @#10

приводит к тому же, что и **CLR 10**, но для ассемблера она равносильна

**CLR @(PC) +
.WORD 10**

(Как транслируется команда **CLR 10**?) В момент вычисления исполнительного адреса **РС** указывает на слово, содержащее число **10**; следовательно, ячейка, которую задает выражение **@(PC)**, имеет адрес **10** — она и очищается командой. Из-за изменения автоинкрементного режима **РС** станет указывать на следующую команду программы.

Абсолютная адресация может применяться и в случае перемещаемых адресов. Так, если **MEM** есть перемещаемая ячейка с адресом **140**, то команда **CLR @#MEM** будет оттранслирована так:

005037
000140'

(Как выглядит результат трансляции команды **CLR MEM**?)

Программа, которая ссылается на фиксированные ячейки памяти (как ячейка **10** в предыдущем примере) с использованием относительной адресации, не будет работать, если ее поместить в произвольное место памяти, — обязательно требуется ее настройка для каждого нового адреса загрузки. Можно, однако, вместо относительной адресации использовать абсолютную, имея в виду, что такие команды, как **CLR @#10**, являются *позиционно-независимыми*.

УПРАЖНЕНИЕ. Является ли команда **CLR @#MEM** позиционно-независимой?

Если в написанной программе применяется относительная адресация, но по каким-то причинам необходимо заменить ее абсолютной, то сделать это можно без большого труда, воспользовавшись удобным приемом, который предусмотрен в ассемблере. Директива

.ENABL AMA

представляет собой предписание ассемблеру трактовать все относительные адреса как абсолютные. Поэтому для смены адре-

сацией достаточно добавить в начало программы всего одну строку и перетранслировать программу. Это может пригодиться во время отладки, потому что при использовании абсолютной адресации легче разбираться в листинге программы. Директива **.ENABL AMA** будет распространяться на все идущие за ней команды, пока ее действие не будет отменено директивой **.DSABL AMA**.

Программно-генерируемые прерывания. Обнаружение несуществующей команды — одно из тех неблагоприятных обстоятельств, при которых процессор прервет программу в ячейке, расположенной в младших адресах памяти. И всякий раз он занесет значения **PS** и **PC** в стек, а новые загрузит из двух слов вектора прерывания.

Этим исчерпываются те случаи, когда прерывание происходит автоматически в виде аппаратной реализации независимо от желаний программиста. Возможно, однако, с помощью специальных команд инициировать прерывание и из программы. В системных программах PDP-11 часто используется команда эмуляции прерывания **EMT** (EMulator Trap). Вектор прерывания команды **EMT** имеет адрес 30. Это означает, что после занесения **PS** и **PC** в стек новые их значения берутся из ячеек 30 и 32. Большинство мониторных вызовов включает эту команду, инициирующую прерывания и, как следствие, вмешательство выполняемой под управлением монитора программы, к которой (по крайней мере в системах с разделением времени) у обычного пользователя нет прямого пути.

Код операции команды **EMT** равен 104000. Хотя в его младшем байте стоит нуль, на самом деле любую команду с тем же самым старшим байтом ассемблер считает командой **EMT**. Это имеет большое значение, поскольку программист получает возможность передавать информацию программе обслуживания прерываний. Код, который попадает в младший байт, задается как операнд в команде **EMT**. Например, команда

EMT 340

транслируется как 104340 и представляет собой в действительности макровывоз **.TTINR** системы RT-11. Эта макрокоманда прочтет литеру с терминала в **R0** и сбросит бит **C**; если же литеры еще нет, бит **C** будет установлен, но ждать ее появления программа не станет. Чтобы дожидаться литеры, нам необходим знакомый уже вызов **.TTYIN**, расширение которого имеет вид

EMT 340
BCS .-2

Точно так же мониторный вызов **.TTOUTR** соответствует команде **EMT 341**, так что знакомый нам вызов **.TTYOUT** имеет расширение

EMT	341
BCS	.-2

Мы не включили сюда команду пересылки литеры из **R0** в ячейку, которая может быть указана в вызове; об этом пойдет речь в § 3.5.

Операционная система в ячейку **30** загрузит адрес программы обслуживания прерываний от команды **EMT**. Эта программа начнет свою работу с сохранения содержимого регистров **R0—R5**, после чего адрес возврата окажется в ячейке **14(SP)**. Он пересылается в **R0**:

MOV 14(SP),R0

Сама команда **EMT** теперь находится по адресу **—2(R0)**. Обслуживающая прерывание программа заносит команду **EMT** в стек:

MOV —(R0), —(SP)

Дальнейшие соображения по этому поводу приводятся в § 4.5. Далее значение младшего байта команды засылается в **R0**:

MOVB (SP)+,R0

Одновременно из стека удаляется теперь уже ненужная команда **EMT** (вспомните, что даже в байтовых командах **SP** увеличивается на 2). То, что получается в результате выполнения двух последних команд, не может быть достигнуто какой-либо одной командой.

Один из способов, который теперь можно применить для передачи управления соответствующей программе, заключается в использовании *n*-переключателя. В ячейке **TABLE** должен начинаться блок из **400** слов, каждое из которых с адресом **TABLE+<2*n>** ($0 \leq n \leq 377$) должно содержать адрес программы **EMT n**. Следовательно, после удвоения содержимого регистра **R0** (для чего?) достаточно простой пересылки:

MOV TABLE(R0),PC

Управление можно передать также командой **JMP**. Ее код операции равен **0001** в четырех левых восьмеричных цифрах слова, при этом на указание приемника остается шесть разрядов. Полный исполнительный адрес будет вычисляться. Таким

образом, вместо команды **MOV** мы могли бы написать

JMP .@TABLE(R0)

Имейте в виду, что команда **JMP**, так же как и команда **JSR**, загружает в счетчик команд **PC** адрес приемника, а не его содержимое. Поэтому, чтобы достигнуть того же результата, что и командой **MOV**, в команде **JMP** требуется поставить метод адресации на один уровень косвенности глубже. Так, команда **JMP (R0)** равнозначна по действию команде **MOV R0,PC**, команда **JMP @(R0)** — команде **MOV (R0),PC**, а команды **JMP**, эквивалентной команде **MOV @(R0),PC**, не существует. Учтите, что форма записи **JMP R0** «некорректна», поскольку в машине PDP-11 не разрешается передача управления на регистр, — в результате произойдет прерывание в ячейке 4 (руководства по некоторым процессорам утверждают, что будет прерывание в ячейке 10; рекомендуем вам непосредственно проверить справедливость этого на вашей машине). Команда **JMP** не изменяет условные признаки.

Имейте также в виду, что команда типа **JMP @(R0)+** загрузит значение **PC** до увеличения содержимого регистра **R0**; то же справедливо и для команды **JSR**. Как это согласуется с нашим представлением о механизме выполнения команды **JSR PC, @(SP)+**?

Программы обслуживания прерываний по команде **EMT** могут употребляться только в программах, работающих без операционной системы, потому что в программном обеспечении фирмы DEC эта команда существенно используется. Вместо нее пользователь может писать команду **TRAP**, которая отличается от команды **EMT** только адресом вектора прерывания. При ее выполнении значение **PC** берется из ячейки 34, а **PS** — из ячейки 36. Код операции равен 104400, причем снова младший байт может быть использован для передачи информации программе обработки.

Вставка заплат. Мы только что рассказали о том, как для вызова подпрограмм можно использовать команду **TRAP** вместо команды **JSR**. В программе должна быть заведена таблица адресов вызываемых подпрограмм, а подпрограммы должны оканчиваться командой **RTI**, а не **RTS**.

Команда **TRAP** экономнее, чем **JSR**, поскольку занимает одно, а не два слова. Это на первый взгляд незначительное преимущество оказывается практически весьма полезным. Не так уж редко случается изменять программу, которая хранится только в виде перемещаемого двоичного файла (типа **.OBJ**) или в виде образа памяти (типа **.SAV**). Существуют системные программы,

предназначенные для облегчения работы по изменению содержимого ячеек и вставки дополнительных участков программы. Последние называются *заплатами*, а работа по их добавлению — *вставкой заплат*. Обычно, однако, у системных программ нет средства перемещения части программы, которое позволило бы освободить место для дополнительных команд. Следовательно, они должны быть занесены в конец файла, а одна из команд заменена командой передачи управления на соответствующую ячейку. Ясно, что гораздо проще найти место для команды **TRAP**, занимающей одно слово, чем для состоящей из двух слов команды **JSR** (конечно, если смещение невелико, то и команда безусловного перехода не менее эффективна). Такой подход осуществим, если только в таблице адресов есть свободное место: поэтому в первой же версии программы неплохо отводить под таблицу как можно больше места в целях ее дальнейшего заполнения.

УПРАЖНЕНИЕ. Разберитесь в программах вашей системы, помогающих вставлять заплаты.

Средства отладки. Команда прерывания для отладки **BPT** (Break Point Trap), имеющая код операции **000003**, вызывает прерывание в ячейке 14. Эта команда не имеет операнда, и поэтому никакой дополнительной информации в обрабатывающую прерывание программу передано быть не может.

Команда **BPT** используется в таких отладчиках, как комплекс **ODT**. Если вам понадобится, чтобы отладчик завел в ячейке **X** вашей программы точку останова, он сохранит адрес и содержимое **X**, а затем выполнит команду

MOV #3,X

заменяя команду, находящуюся в этой ячейке, на команду **BPT**. Когда в процессе исполнения будет достигнут адрес **X**, произойдет прерывание и управление будет передано обслуживающей его программе, входящей в комплекс **ODT**. На этом этапе программист может давать указания отладчику. Если ему будет приказано продолжить исполнение программы, то он восстановит команду по адресу **X**, проверит адрес возврата в стеке (что именно необходимо проверить?) и осуществит выход командой **RTI**.

После достижения точки останова часто бывает желательно посмотреть, как исполняется каждая команда в отдельности. Отладчик предоставляет возможность такого пошагового исполнения, но осуществляет это без использования команды **BPT**, — действительно, было бы слишком громоздко на каждом шаге засылать и удалять эту команду. Вместо этого **ODT** использует другой способ инициации прерывания в ячейке 14: он устанавли-

вает бит *T* — четвертый разряд *PS*. Бит *T* устанавливается, если операнд команды, которая загружает новое содержимое *PS*, содержит 1 в четвертом бите. Поэтому ответ отладчика на запрос пошагового исполнения программы может заканчиваться командами

BIS #20,2(SP)
RTI

Аппаратура устроена так, что, хотя команда **RTI** в данный момент и установила бит *T*, процессор этого еще не «замечает», и поэтому он, как и обычно, продолжает выполнение следующей команды. Поскольку команда **RTI** вернула управление программе пользователя, это именно та команда, которая должна быть исполнена в пошаговом режиме. Команда выполняется, и теперь ЦП «замечает», что бит *T* в *PS* установлен, и осуществляет прерывание в ячейке 14. Оно называется *трассировочным прерыванием*.

УПРАЖНЕНИЯ. 1. Должен ли отладчик, когда ему возвращается управление после трассировочного прерывания, что-либо изменять в ячейке 2(*PS*)?

2. Какие команды, кроме **RTI**, могут устанавливать или сбрасывать бит *T*?

3. Что происходит, когда выполняющаяся в пошаговом режиме команда устанавливает бит *T*?

3.5. Структурная разработка программы

В этом параграфе мы рассмотрим некоторые возможности ассемблера, облегчающие структуризацию программы.

Макрокоманды. Как было отмечено в § 3.3, в системе команд PDP-11 отсутствует байтовый вариант команды **ADD**. Конечно, не представляет особого труда написать собственную программу сложения чисел, расположенных в байтах. Предположим, что нам хочется произвести операцию, которая выполнялась бы так:

ADDB X,Y

если бы такая команда существовала. Разумное решение состоит в том, чтобы переслать содержимое байтов ячеек *X* и *Y* в промежуточные ячейки, затем произвести сложение последних и заслать байтовую часть результата в *Y*.

Однако это не совсем то, что требуется, поскольку не устанавливаются правильные условные признаки. Далеко не всегда программист твердо знает, что ни при каких обстоятельствах

сложение не приведет к переполнению. Поэтому моделирующая команду **ADDB** программа не должна лишать нас возможности проверить бит *V*. Если исходные данные пересылаются в младшие байты промежуточных ячеек, то состояния условных признаков не будут правильно отражать результат последующего сложения. Чтобы получились правильные признаки, прежде всего нужно заслать данные в старшие байты рабочих ячеек, а младшие их байты очистить:

```

                                MOVB      X,L1+1
                                MOVB      Y,L2+1
                                ADD        L1,L2
                                ...
                                ...
L1:                            .WORD      0
L2:                            .WORD      0

```

В данном случае очень неудобно проводить вычисления в регистрах (но все-таки для практики напишите соответствующую последовательность команд, не забывая о том, что при работе с регистром команда **MOVB** размножит знак).

Команда **ADD** оставляет результат в ячейке **L2+1** и корректно устанавливает признаки. Но нельзя переслать результат в **Y**, не сбросив при этом бит *V*! Убедитесь в том, что другие признаки команда **MOVB L2+1,Y** не изменяет. Поэтому потребуется маленькая хитрость. Воспользуемся командой **BVS**, и если в этот момент бит *V* оказался равным 1, то уже *после* пересылки байта данных в **Y** выполним команду **SEV**.

УПРАЖНЕНИЕ. Завершите программу «**ADDB**».

Хотя, быть может, и полезно иметь фрагмент, выполняющий отсутствующую команду **ADDB**, но вставлять его во все те места программы, где требуется подобное вычисление, — занятие не из приятных. Конечно, его можно оформить как подпрограмму, но

```

                                JSR        R5,ADDB
                                .WORD      X,Y

```

выглядит не столь привлекательно, как

```

                                ADDB      X,Y

```

Как упоминалось в § 1.4, ассемблер позволяет программистам создавать собственные команды или *макро* и вызывать их с помощью одного предложения языка ассемблера. Мы уже знакомы с несколькими примерами системных макрокоманд; благодаря директиве **.LIST ME** вы видели, что ассемблер *расширяет*

макрокоманды до их мнемокодной записи. Сейчас мы рассмотрим, как написать макро **ADDB**, чтобы команду

ADDB X,Y

можно было включать в программу наравне с обычными командами языка ассемблера.

Но такому *макровывозу* в программе должно предшествовать *определение* этого макро (*макроопределение*). Оно начинается с директивы **.MACRO**, которая в нашем случае выглядит так:

.MACRO ADDB X,Y

где в качестве аргументов директивы записывается сначала выбранное нами имя макро, а потом список переменных, с которыми данный фрагмент оперирует. Между переменными, а также между именем макро и переменной должен стоять *разделитель*. Разделителями могут служить пробелы, символы табуляции и запятые. Используя запятые для разделения переменных, а символ табуляции для отделения имени макро от переменной, мы получаем директиву **.MACRO**, согласующуюся с формой записи макровывозов, которые далее встретятся в программе. В нашем случае фрагмент оперирует с двумя байтами, содержимое которых складывается и которые мы обозначили **X** и **Y**.

Вслед за директивой **.MACRO** в макроопределении идет последовательность команд (*тело макро*), которую в программе должен представлять макровывоз. Здесь это как раз та последовательность команд, которую вы написали, когда выполняли предыдущее упражнение. Макроопределение должно заканчиваться директивой **.ENDM**, по которой ассемблер узнает о конце макроопределения. В директиве **.ENDM** можно в качестве параметра указать имя макро

.ENDM ADDB

Это поможет вам ориентироваться при написании программы.

Так, определение системной макрокоманды **.TTINR**, которая упоминалась в § 3.4, имеет вид

**.MACRO .TTINR
 EMT 340
.ENDM**

Отсюда видно, что **.TTINR** есть фрагмент, который читает литературу в заранее выбранную ячейку — регистр **R0** (об этом заботится команда **EMT**), поэтому дополнительных параметров в директиве **.MACRO** нет.

Параметры макро. В макро **ADDB** есть параметры: ячейки **X** и **Y**. Важно понять ту роль, которую играют эти символы, называемые *формальными параметрами*.

Функция формальных параметров заключается в установлении синтаксиса макровывода.

Директива **.MACRO** сообщает ассемблеру, что в макровыводе **ADDB** будет два параметра и что в теле макро **X** «обозначает» первый параметр в макровыводе, а **Y** — второй.

Допустим теперь, что где-то в программе, но обязательно после определения макро **ADDB** есть строка

```
LABEL:      ADDB      @MEM,WRD+1
```

где **MEM** и **WRD** — ячейки, определенные в каком-то другом месте программы. Ассемблер соотнесет **@MEM** — первый параметр в макровыводе с **X** — первым формальным параметром в директиве **.MACRO**. В результате при расширении макровывода с меткой **LABEL** он заменит все ссылки на имя **X** в теле макро на ссылку **@MEM**. Поскольку первой строкой тела макро была команда **MOVB X,L1+1**, первая строка расширения выглядит так:

```
LABEL:      MOVB      @MEM,L1+1
```

Аналогично имя **Y**, где бы оно ни встретилось в теле макро, ассемблер заменит на **WRD+1**. Поэтому следующая за меткой строка будет такой:

```
MOVB      WRD+1,L2+1
```

Параметрами макровывода могут быть любые выражения, но только «понятные» ассемблеру.

Обратите внимание на то, что в самом макровыводе мы не употребляем имена, выбранные в качестве формальных параметров, — все подстановки производятся ассемблером автоматически. Действительно, в макровыводе **ADDB** нельзя ссылаться на **X** и **Y**, поскольку ячейки с соответствующими именами не были заведены в нашей программе. Ассемблеру нет никакого смысла отводить под формальные параметры место в памяти, поэтому вне макроопределения имена **X** и **Y** никакого значения не имеют. Отсюда следует, что без какой бы то ни было двусмысленности они могут употребляться в нескольких макроопределениях, а также, как и обычно, в качестве имен ячеек памяти внутри самой программы, если только правильно в ней определены.

УПРАЖНЕНИЕ. Оформите фрагмент **ADDB** в виде макроопределения. Напишите программу, в которой есть макровывоз **ADDB**. Сравните листинги этой программы, когда в ней есть директива **.LIST ME** и когда ее нет.

Макроимена. Макро можно вызывать в программе не один раз. И всякий раз ассемблер будет выполнять одну и ту же процедуру, заменяя макровывоз на тело макро. Формальные параметры при этом будут заменены на параметры данного вызова. Таким образом, одно из возможных определений системной макрокоманды **.TTYIN**, позволяющее занести литеру в произвольно выбранную ячейку, могло бы быть таким:

```
.MACRO      .TTYIN      X
              EMT        340
              BCS        .-2
              MOV        R0,X
.ENDM
```

Тогда расширение макровывозов

```
.TTYIN      MEM
...
...
.TTYIN      WRD
```

выглядело бы так:

```
EMT        340
BCS        .-2
MOV        R0,MEM
...
...
EMT        340
BCS        .-2
MOV        R0,WRD
```

Заметьте, что если мы в программу включаем макроопределение **.TTYIN**, то необходимость в директиве **.MCALL** отпадает.

И вот теперь, желая неукоснительно следовать высказанному ранее совету и поэтому избегать в командах перехода выражений, включающих точку, мы заново перепишем макроопределение:

```
.MACRO      .TTYIN      X
L1:          EMT        340
              BCS        L1
              MOV        R0,X
.ENDM
```

Последовательность макрокоманд **.TTYIN** приведет к появлению двух различных ячеек с одной и той же меткой **L1**, что вызовет

ошибку, когда ассемблер попытается транслировать второе из них.

Ассемблер справился бы с задачей создания различных имен меток при каждом макровывозе, если только ему сообщать, с какими именно метками он должен так поступать. Метки, для которых необходимо создавать такие *локальные имена*, должны быть объявлены в качестве параметров директивы **.MACRO**, причем перед каждым из них должна стоять литера ?. Для удобства чтения желательно отделять их от «настоящих» формальных параметров табуляцией:

```
.MACRO      .TTYIN      X      ?L1
```

Вот и все, что должен сделать программист. Не нужно ничего изменять ни в теле макро, ни в форме макровывоза.

Ассемблер будет создавать для таких меток локальные имена, используя уже знакомую нам форму *n\$*, но начиная с 64\$. Заметим, что, как и в локальных метках, задаваемых программистом, числа, предшествующие символу \$, являются десятичными.

На рис. 3.7 представлен листинг фрагмента программы, содержащий действующую версию макро **ADDB** вместе с двумя малоправдоподобными вызовами, приведенными лишь для иллюстрации способа подстановки параметров. Заметьте, что, поскольку в параметр **<WRD—MEM>&"ZX** входят угловые скобки, весь собственно параметр должен быть заключен в еще одну пару угловых скобок — в противном случае закрывающая скобка **>** будет рассматриваться ассемблером как признак конца параметра, а символы **&"ZX** — как неправильная метка, которая должна быть поставлена вместо метки **L1**.

Обратите внимание, что нумеруются только те строки программы, которые соответствуют командам, написанным программистом, но не командам, порожденным ассемблером. Поскольку само по себе макроопределение команд не порождает, его можно располагать в любом месте программы, лишь бы только оно предшествовало первому макровывозу.

Обратите внимание на то, как нам удалось локальной макрометкой пометить следующую за макровывозом команду. При этом вспомните, что ассемблер игнорирует пустые строки, как видно из второго столбца листинга.

УПРАЖНЕНИЯ. Напишите макроопределения для выполнения следующих операций:

а) **SWAP X, Y** — обмен содержимым между ячейками **X** и **Y**; напишите также байтовую версию.

б) **FILCMP X, Y** — поэлементное сравнение двух блоков, начинающихся в **X** и **Y** и заканчивающихся первым нулевым эле-

ментом. Если оба блока идентичны по длине и хранящейся в них информации, макрокоманда должна удалить весь блок, начинающийся в Y.

в) **IREAD N, Y** — чтение N целых чисел с терминала в блок памяти с начальным адресом Y.

MACTST MACRO V03.01 5-JAN-79 09:52:22 PAGE 1

1					.TITLE	MACTST
2					.LIST	ME
3				.MACRO	ADDB	X,Y ?L1,?L2,?L3,?L4
4					MOVB	X,L1+1
5					MOVB	Y,L2+1
6					ADD	L1,L2
7					BVS	L3
8					MOVB	L2+1,Y
9					BR	L4
10			L1:		.WORD	0
11			L2:		.WORD	0
12			L3:		MOVB	L2+1,Y
13					SEV	
14			L4:			
15			.ENDM			
16						
17	000000			START:	ADDB	@MEM,<<WRD-MEM>&"ZX>
	000000	117767	000124		MOVB	@MEM,64\$+1
	000006	116767	000002		MOVB	<WRD-MEM>&"ZX,65\$+1
	000014	066767	000014		ADD	64\$,65\$
	000022	102406			BVS	66\$
	000024	116767	000007		MOVB	65\$+1,<WRD-MEM>&"ZX
	000032	000406			BR	67\$
	000034	000000		64\$:	.WORD	0
	000036	000000		65\$:	.WORD	0
	000040	116767	177773	66\$:	MOVB	65\$+1,<WRD-MEM>&"ZX
	000046	000262			SEV	
	000050			67\$:		
18	000050				ADDB	MEM-WRD,@MEM
	000050	116767	177776		MOVB	MEM-WRD,68\$+1
	000056	117767	000046		MOVB	@MEM,69\$+1
	000064	066767	000014		ADD	68\$,69\$
	000072	102406			BVS	70\$
	000074	116777	000007		MOVB	69\$+1,@MEM
	000102	000406			BR	71\$
	000104	000000		68\$:	.WORD	0
	000106	000000		69\$:	.WORD	0
	000110	116777	177773	70\$:	MOVB	69\$+1,@MEM
	000116	000262			SEV	
	000120			71\$:		

Рис. 3.7. Листинг макро **ADDB**.

г) **PRINT X** — выдача на терминал записанного в побайтовой форме в коде ASCII текста с началом в ячейке X, конец которого определяется по первому встреченному нулевому байту. Решите сами, какая форма вызова для распечатки текста, расположенного с адреса MEM, вам больше подходит: **PRINT MEM** или **PRINT #MEM**.

В упражнениях в) и г) вам придется использовать *вложенные* макро. В макроопределении **IREAD** будет системная макрокоманда **.TTYIN**, а в макроопределении **PRINT** — макрокоманда **.TTYOUT**. Вложенность макровывозов неограниченна, если

только каждому макровывозу предшествует соответствующее макроопределение.

Попутно рис. 3.7 показывает, что при помощи длинных макро очень просто порождать необозримо длинные программы. Этого можно избежать, если писать в макротеле только последовательность вызова подпрограммы, которая выполняет необходимые действия (почему такой способ разрешает проблему?):

```
.MACRO      ADDB
             JSR      R5,XADDB
             .WORD    X,Y
.ENDM
```

По нашему мнению, подпрограмме и макрокоманде лучше давать различные имена (здесь к имени подпрограммы добавлена буква **X**), но ассемблер этого не требует. Учтите, что в системных макро используется именно такая методика: макро представляет собой просто последовательность вызовов для команды **EMT**.

Выбор основания системы счисления. В макрокомандах, которые предполагается использовать в различных программах, необходимо заботиться о том, чтобы ничто в программе не могло помешать выполнению их функций. Системные макрокоманды защищены от директивы ассемблера **.RADIX**, параметром которой служит одно из десятичных чисел 2, 4, 8, 10 и которая приводит к тому, что встречающиеся во всех последующих командах числа ассемблер рассматривает как заданные в системе счисления с установленным основанием. Так, последовательность

```
.RADIX      10
CMP          #1000,MEM
```

есть сравнение содержимого **MEM** с 1000 в десятичной системе счисления.

Если бы перед мониторным вызовом **.TTYIN** встретилась директива **.RADIX 10**, то команда **EMT 340** была бы оттранслирована с десятичным параметром 340, что привело бы к неверной передаче управления. В этой макрокоманде следует воздержаться от установки основания восьмеричной системы счисления (ради одного только числа 340 в команде **EMT**). Символ **^** (стрелка вверх, но не **CONTROL**), за которым следует буква, задает основание для последующего числа (или выражения, заключенного в угловые скобки). Можно использовать буквы **B** (binary — двоичное основание), **O** (octal — восьмеричное) или **D** (decimal — десятичное).

```
EMT          ^O340
```


Условная трансляция. Макроопределение **.TTYIN** в системе RT-11 имеет вид

```

      .MACRO      .TTYIN      CHAR
                      EMT      ^O340
                      BCS      .-2
      .IF NB <CHAR>
      .IF DIF <CHAR>, R0
                      MOV B    %0,CHAR
      .ENDC
      .ENDC
      .ENDM

```

(в разных версиях системы возможны небольшие различия). Как видно, команда **MOV B %0,CHAR** с обеих сторон окружена двумя директивами ассемблера. Их функция состоит в отмене трансляции этой команды, когда в ней нет необходимости.

В директиве **.IF** задается условие, при выполнении которого трансляция должна иметь место, а далее следует выражение (или выражения), которое необходимо проверять. Условие и выражение могут быть разделены одним или несколькими пробелами, символом табуляции или запятой. Выражения отделяются запятыми.

Здесь у нас имеются две условные директивы **.IF**, предназначенные специально для работы с параметрами макрокоманды. Имя такого параметра необходимо заключать в угловые скобки.

Директива **.IF NB** сообщает ассемблеру, что при обработке макровызова он должен транслировать последующие команды только в том случае, если параметр вызова не пуст (Non Blank). Параметр в обращении **.TTYIN MEM** присутствует (что соответствует непустому значению параметра **CHAR**), поэтому условие **.IF NB <CHAR>** удовлетворяется, и ассемблер переходит к следующей строке, о которой мы поговорим попозже.

Допустим, однако, что вызов имеет вид **.TTYIN**. Здесь поле параметра, соответствующего **CHAR**, пусто, вследствие чего ассемблер не станет транслировать дальнейший текст до тех пор, пока не встретит директиву **.ENDC** (END Conditional), которая является парной к встретившейся директиве **.IF**. В приведенном примере директивы **.IF** оказались вложенными, и каждой из них соответствует своя директива **.ENDC**, подобно парам открывающих и закрывающих скобок. Директиве **.IF NB** соответствует вторая из директив **.ENDC**, а директиве **.IF DIF** — первая.

Таким образом, результат трансляции вызова **.TTYIN** таков:

```

      EMT      ^O340
      BCS      .-2

```


Напомним, что эта программа **ЕМТ** читает литеру в регистр **R0**, и, следовательно, вызов **.TTYIN** интерпретируется как **.TTYIN R0** и кодируется без оказывающейся излишней команды **MOVB R0, R0**.

Но команда **MOVB** является лишней не только при вызове **.TTYIN**, но и при вызове **.TTYIN R0**. Поле аргумента здесь не пусто; следовательно, директива **.IF NB** не приведет к отказу от трансляции команды **MOVB**. Эту функцию выполняет директива **.IF DIF** (**IF DIF**ferent — если различны), разрешающая ассемблеру транслировать следующий ниже блок, если только два выражения, заданные в ней в качестве параметров, различны. Можно сравнивать два параметра макрокоманды или параметр макрокоманды с выражением (о чем упоминалось выше). Согласно руководствам, здесь снова каждый макропараметр должен заключаться в угловые скобки. Однако нам встретилось несколько ассемблеров, для которых это оказалось делать не обязательно.

Так, предположим, что мы решили написать макро **SWAP** для обмена содержимым между двумя ячейками:

```
.MACRO      SWAP      X,Y
```

Если в макровывозе одна и та же ячейка указана в качестве обоих параметров

```
SWAP MEM, MEM
```

то в каких-либо действиях нужды нет. Мы можем либо положиться на здравый смысл будущих пользователей макрокоманды, который убережет их от такой формы вызова, либо запретить ассемблеру предпринимать что-либо, начав тело макро директивой

```
.IF DIF <X>,<Y>
```

Конец данного условия (директива **.ENDC**) должен непосредственно предшествовать завершающей макроопределение директиве **.ENDM**.

Если для обмена содержимым **X** и **Y** кратковременно используется регистр **R0**, то в макроопределение можно включить такие команды:

```
MOV      X,R0
MOV      Y,X
MOV      R0,Y
```


или, стремясь к максимальной экономии, такие:

```
.IF DIF      <X>,R0
              MOV      X,R0
,ENDC
              MOV      Y,X
.IF DIF      <Y>,R0
              MOV      R0,Y
,ENDC
```

Если условие записывается в одну строку, то можно применить директиву **.IIF** (Immediate IF); тогда директива **.ENDC** не нужна. Теперь все макроопределение принимает вид

```
.MACRO      SWAP      X,Y
.IF DIF      <X>,<Y>
.IIF DIF      <X>,R0    MOV      X,R0
              MOV      Y,X
.IIF DIF      <Y>,R0    MOV      R0,Y
,ENDC
.ENDM
```

Имеется противоположная по условию к директиве **.IF DIF** директива: транслировать, если параметры идентичны **.IF IDN** (IF arguments are IDeNtical). Дополнительной к **.IF NB** является директива **.IF B** (IF Blank — если пуст). Во всех этих условиях параметры макрокоманды рассматриваются просто как цепочка литер. Директивы **.IF B** и **.IF NB** проверяют, есть ли вообще литеры в цепочке. В директивах же **.IF IDN** и **.IF DIF** сравнивается, состоят ли цепочки из одних и тех же литер. Так, например, если **MEM** — перемещаемая ячейка **100**, а **WRD** — перемещаемая ячейка **200**, то в результате вызова **SWAP MEM+100,WRD** в макрорасширение попадут все три команды. Как цепочки литер параметры **MEM+100** и **WRD** совершенно различны, хотя как выражения оба они равны перемещаемому адресу **100**.

УПРАЖНЕНИЕ. Как будет выглядеть расширение макро-вызовов

```
(a) .TTYIN %0
(б) REG0=%0
    .TTYIN REG0
```

Условия, включающие выражения. При помощи директивы **.IF NE** (IF Not Equal to zero — если не равно нулю) можно дать указание на трансляцию при условии несовпадения параметров как выражений:

```
.IF NE X-Y
```


Заметьте, что проверяемое в условии выражение не нужно заключать в угловые скобки. Имейте также в виду, что в директиву **.IF NE** входит только одно выражение, которое сравнивается с нулем.

В нашей макрокоманде **SWAP** вместо сравнения **.IF DIF <X>, <Y>** мы могли бы поставить условие **.IF NE X—Y**. Будем опять считать, что **MEM** соответствует перемещаемому адресу 100, а **WRD** — перемещаемому адресу 200. Теперь уже вызов

SWAP MEM + 100,WRD

очевидно, содержит равные параметры. В то же время в макро-вызове

SWAP WRD — MEM, MEM

они различны: первый параметр равен абсолютному числу 100, а второй — перемещаемому. (Что произойдет, если такой макро-вызов встретится в абсолютной секции программы?) В процессе трансляции такого вызова (с перемещаемым адресом **MEM**) ассемблер выдаст сообщение об ошибке, поскольку при вычислении будет фигурировать запись **WRD — MEM — MEM**, которая не является ни абсолютной, ни перемещаемой и потому в качестве выражения не допускается.

Учтите, что вызов **SWAP MEM, R0** также приведет к ошибке, так как ассемблер будет пытаться вычислить разность **MEM — R0**, а поскольку регистры в PDP-11 не имеют адресов, такая запись не допускается в качестве выражения.

Существуют следующие директивы, в которых значение выражения сравнивается с нулем и проверяется: отлично ли оно от нуля (**NE**); больше (**GT**); меньше или равно (**LE**); меньше (**LT**); больше или равно (**GE**).

Естественно, что не существует способа, при помощи которого ассемблеру можно дать директиву транслировать что-либо в зависимости от значения содержимого некой ячейки (почему?).

Переходы при условной трансляции. Не всегда бывает желательно расширять часть макро в случае выполнения условия и пропускать ее в противном случае. Иногда нам хотелось бы предпринять альтернативные действия в зависимости от того, удовлетворяется условие в директиве **.IF** или нет.

Допустим, к примеру, что нам понадобилась макрокоманда **STORE** с одним параметром, причем при вызове **STORE MEM**, где **MEM** — ячейка памяти, она должна загружать адрес **MEM** в стек: **MOV #MEM, — (SP)**, а при вызове **STORE R**, где **R** — регистр, заносить в стек его содержимое: **MOV R, — (SP)**. Та-

кую макрокоманду можно было бы использовать внутри другой макрокоманды для работы в зависимости от определенных обстоятельств с параметрами последней.

Сначала мы должны распознать, является ли параметр регистром. Включенная в тело макродиректива **.NTYPE** сообщит нам номер режима и номер регистра, которые используются при адресации к любому параметру. Эта директива имеет два параметра: имя, которое мы хотим идентифицировать с сообщаемой директивой **.NTYPE** информацией, и параметр, которым мы интересуемся. Можно выбрать любое имя согласно обычным правилам; ему присваивается шестизначный код, определяющий режим адресации и регистр, точно так же, как и в операторе прямого присваивания. Итак, после команд

```
.MACRO      STORE      X
            .NTYPE      A,X
```

когда в условных директивах мы будем ссылаться на имя **A**, его значение будет равно восьмеричному числу в диапазоне от 0 до 77 в соответствии со значениями шести битов, задающих способ адресации и номер используемого регистра того фактического параметра, который будет фигурировать вместо **X** в макровыводе.

Этот параметр будет регистром в том и только в том случае, когда номер режима адресации равен нулю, т. е. когда директива **.NTYPE** передаст значение, заключенное между 0 и 7. Следовательно, после

```
            .NTYPE      A,X
        .IF EQ A&70
```

последующие команды будут транслироваться, только если в качестве параметра макровывода был употреблен регистр. Поэтому следующая строка в теле должна быть **MOV X,—(SP)**. Теперь мы можем написать все макроопределение:

```
.MACRO      STORE      X
            .NTYPE      A,X
        .IF EQ A&70
            MOV          X,—(SP)
        .IFF
            MOV          #X,—(SP)
        .ENDC
        .ENDM
```

Смысл директивы **.IFF** таков: продолжать транслировать, если результат проверки условия, задающего текущий блок условной трансляции, оказался отрицательным. Обратите внимание на то, что директива **.IFF** не является границей нового блока;

напротив, она *подчинена* условной директиве **.IF EQ**, и указывает, что должно быть оттранслировано, если условие не удовлетворяется.

УПРАЖНЕНИЯ. 1. Перепишите макроопределение **STORE**, используя вместо **.IF EQ** директиву **.IF NE**.

2*. Напишите заново макроопределение **STORE** так, чтобы к его параметру можно было адресоваться любым способом.

3. Напишите макроопределение **BMOV**, которое выполняло бы в точности такую же операцию, как и **MOVB**, за исключением того, что если параметром является регистр, то его старший байт будет оставаться без изменения.

4. Напишите макро для выполнения операции **AND** над содержимым двух ячеек памяти.

Ссылки вперед. Если в макрокоманде встречаются ссылки на ячейку памяти, расположенную за ней, то могут возникнуть трудности. Что это за трудности и каким образом их преодолевать, зависит от используемой версии ассемблера, так что здесь возможны всякие неожиданности.

В макро может потребоваться выполнить некоторые действия только при первом ее вызове внутри программы. Например, может понадобиться отвести место для временного хранения информации. Обычный способ состоит в применении директивы **.IF DF** (**IF DeFined**), которая разрешает дальнейшую трансляцию только в том случае, если ее параметр был определен. Противоположная к ней директива — это **.IF NDF**. Имя, выбранное в качестве формального параметра, не должно встречаться где-либо в другом месте программы. Так, макро **STARTUP** будет вызвано только один раз, если в основную макро включить

```
.IF NDF      Q
              Q=0
              STARTUP
.ENDC
```

При первом вызове основного макро значение **Q** не определено, но внутри этого вызова оно полагается равным нулю, и, следовательно, при следующем вызове директива **.IF NDF** прекратит трансляцию.

Однако в некоторых версиях макроассемблера **MACRO-11** еще до стадии принятия ассемблером решений по условной трансляции «обращается внимание» на включенный в тело макрокоманды оператор прямого присваивания **Q=0** (и имя **Q** включается в таблицу распределения имен). В результате имя **Q** будет определено даже при первом макровывозе, и потому макро **STARTUP** вообще никогда не будет вызвана.

Для таких трансляторов имя **Q** можно определить внутри макро, которую программа должна найти в библиотеке. Так макро **.MACS**, входящая в систему **RT-11**, выполняет некоторые действия, необходимые другим системным макрокомандам, и, в частности, включает выражение прямого присваивания **...V1=3**. В других системных макро встречается запись

```
.IF NDF      ...V1
              .MCALL      .MACS
              .MACS
.ENDC
```

Вопрос о том, определено имя или нет, возникает и в несколько ином плане. В качестве учебной очень популярна задача написания макро для генерации команды **BR**, если адрес ячейки-параметра лежит внутри области действия этой команды, и команды **JMP**, если—нет:

```
.MACRO      J      X
  .IF условие
            BR      X
  .IFF
            JMP     X
.ENDC
.ENDM
```

(Каким должно быть *условие*?) Допустим сначала, что мы имеем в программе

```
LABEL:      ...
            ...
            ...
            J      LABEL
```

Здесь пока трудности нет: ассемблер проверяет, достаточно ли мала величина выражения **LABEL—.**, чтобы можно было поставить команду **BR**, и поступает в соответствии с результатом.

Предположим, однако, что у нас иная ситуация:

```
            J      LABEL
            ...
            ...
LABEL:      ...
```

Ассемблер может отреагировать двояким образом в зависимости от используемой версии. Если он устроен так, что должен раскрыть макровывоз до того, как дойдет до метки **LABEL** и занесет ее в таблицу имен, то в приведенном примере **LABEL** будет считаться неопределенным именем. Это будет трактоваться как ошибка. При таком ассемблере рассмотренный выше прием с введением имени **Q** должен сработать.

Ассемблер, устроенный по-другому, может попытаться узнать, определена ли метка **LABEL** или нет. Но поскольку он еще не закодировал команды, расположенные между текущим значением счетчика адресов и меткой **LABEL**, ему недоступно значение адреса последней. Поэтому на этом этапе он не сможет вычислить адресное поле команды **JMP** или смещение для команды **BR**. Более того, он не сможет даже определить, какая из этих команд требуется.

Существует несколько методик разработки ассемблеров (обсуждение которых выходит за рамки данной книги), позволяющих разрешить рассмотренную проблему ссылок вперед в макрокомандах. В результате применения таких методик в макрокоманде можно ссылаться вперед на метку командой **MOV** (как мы уже это видели при разработке макрокоманды **ADDB**), командой ветвления и т. д. Применение команды **JMP**, однако, на разных стадиях трансляции приведет к получению различных значений величины смещения; такая ситуация носит название *ошибка фазы трансляции* и обозначается в листинге буквой **P**.

4. ПЕРИФЕРИЙНОЕ ОБОРУДОВАНИЕ

4.1. Ввод-вывод с терминала и пульта управления

До сих пор задачу обмена данными между вычислительной машиной и внешними устройствами мы возлагали на операционную систему, используя вызовы монитора **.TTYIN**, **.TTYOUT** и **.PRINT**. Напомним, что во всех случаях расширение этих системных макро содержит команду **ЕМТ**, которая инициирует вызов специальной подпрограммы, обрабатывающей операции ввода-вывода. В системе разделения времени рядовому пользователю приходится полагаться на управляемые монитором программы, поскольку такие системы не предоставляют пользователям, не имеющим особых привилегий, прямого доступа к механизму ввода-вывода.

Однако в изолированной системе прямой доступ к вводу-выводу возможен. Это и будет предметом обсуждения данного и следующего параграфов. И независимо от того, есть ли в вашем распоряжении такие средства или их нет, эти разделы следует изучить в полном объеме, чтобы углубить представление о структуре системы PDP-11.

Обратимся к нашей самой первой программе и научимся печатать букву **В** на терминале, но на этот раз без помощи монитора. Цель программы теперь состоит в том, чтобы передать данные туда, откуда терминальное устройство сможет их забрать. Это — электронное запоминающее устройство, находящееся в терминале. Оно называется *терминальным буфером вывода*. Терминал интерпретирует содержимое буфера как код ASCII и преобразует его в соответствующие литеры, которые печатаются на бумаге или высвечиваются на экране дисплея.

Терминальный буфер вывода имеет свой адрес, так же как и обычная ячейка памяти. Данные в эту ячейку можно заносить обычными командами языка ассемблера с любой адресацией. Адрес терминального буфера фиксирован и равен **177566**. Для прямого занесения в буфер вывода с пультовых переключателей (речь о них пойдет чуть ниже) вам, возможно, придется набрать на них адрес **777566** или даже **17777566**. Но в программах адресом буфера вывода остается **177566** (даже в программах, загружаемых через переключатели). Причина такого несоответствия будет рассмотрена в § 4.5.

Буфер содержит восемь доступных программисту битов, и к нему нужно адресоваться как к байту. Следовательно, наша программа будет такой:

```
                                TPB=177566
START:    MOVB      #102,TPB
                                HALT
                                .END      START
```

Мнемоника **TPB** является стандартной для терминального буфера вывода.

Поскольку в этом параграфе монитор для нас не существует, нам придется остановить ЦП командой **HALT**, а не передавать управление монитору через **.EXIT**. Такую простую программку можно загрузить и исполнить даже без помощи операционной системы.

Пульты переключатели. Большинство моделей PDP-11 имеет пульт оператора, снабженный лампочками и переключателями, посредством которых можно управлять системой. Попробуем исполнить предыдущую программу, используя пульты переключатели. Поскольку ассемблера нет, мы должны закодировать ее сами. В процессе кодирования не стоит забывать, что загрузчика тоже нет. Поэтому, чтобы не создавать себе дополнительных забот по перемещению программы, следует писать позиционно-независимые команды. Тогда без команд ассемблера наша программа и соответствующий ей восьмеричный код выглядят так:

```
112737                                MOVB      #102,@#177566
000102
177566
000000                                HALT
```

Обычно операционная система загружает программы пользователя с адреса **1000**. Имея в своем распоряжении всю машину, мы вольны размещать программу с любого адреса. Тем не менее, чтобы избежать пересечения с областью размещения различных векторов прерывания и чтобы оставить место под стек, мы также будем начинать загрузку с тысячной ячейки.

Включаем машину (только не установкой тумблера включения в положение «блокировка», что блокирует пульты переключатели) и набираем на переключателях пульта число **1000**. Переключатели, перенумерованные справа налево, начиная с нуля, соответствуют битам в слове (в наиболее сложных моделях PDP-11 используется восьмеричная индикация). Чтобы высветить **1000** (восьмеричное), достаточно поднять переключатель

номер 9 (десятичное), включая таким образом лампочку соответствующего разряда. Включаем теперь загрузку адреса (будем обозначать этот переключатель буквой **L**) и проверяем значение адреса 1000 в двоичном виде на индикационном регистре адреса (горящая лампочка соответствует 1).

Продолжаем загрузку программы. При помощи переключателей набираем 112737 — первое слово программы и затем поднимаем переключатель **D**, соответствующий занесению. Новое содержимое ячейки 1000 появится на лампочках индикационного регистра данных.

Для переключателя **D** предусмотрен автоматический пошаговый режим. Его включение приводит к занесению информации по текущему адресу и увеличению регистра адреса на 2. Поэтому надо установить на переключателях следующее слово и поднять **D** и т. д. Когда загрузка закончена, можно воспользоваться переключателем **L**, чтобы вновь установить на регистре адреса 1000 и проверить коды с помощью тумблера «чтение» (**E**). Включение этого тумблера также автоматически увеличивает регистр адреса.

Остается только вновь при помощи **L** задать начальный адрес программы, проверить, что тумблер остановки процессора не находится в состоянии **HALT**, и включить «пуск» (**S**). Команды начнут выполняться с 1000-го адреса, будет напечатана буква **B**, и процессор остановится.

Некоторые пульта снабжены специальным комплектом переключателей для индикации содержимого регистров и занесения в регистры. Обычно регистры находятся в диапазоне «адресов» с 177700 для **R0** по 177707 для **PC**. Обратите внимание на такую особенность: следующий регистр становится доступным, если адрес регистра увеличивается на единицу. Переключатели **D** и **E** обеспечивают автоматический переход к следующему регистру лишь на некоторых типах ЦП. Заметьте также, что эти «адреса», если они встречаются в программе, совершенно бессмысленны.

Не на всех машинах PDP-11 имеются пульты переключателей. Некоторые примеры, рассмотренные в этой главе, были проверены на машине PDP-11/04 с помощью *эмулятора пульта управления*. Он представляет собой устройство, которое включается нажатием **BOOT** (смысл этого названия станет ясен позднее) на машине 11/04. Эмулятор пульта управления — это программа, которая не загружается в оперативную память, а все время находится в постоянном запоминающем устройстве (ПЗУ).

Эмулятор устанавливает связь между процессором и терминалом, а затем принимает команды, имитирующие различные функции переключателей пульта. При готовности к вводу очередной команды эмулятор модели 11/04 печатает символ \$. Чтобы загрузить рассмотренную ранее программу, пользователь

должен набрать **L 1000** ←| и дождаться очередного приглашения, затем набрать **D 112737** ←| и потом после нового \$ набрать **D 102** ←|, опуская нули в начале числа, и т. д.

Буква **E** с пробелом (но не возвратом каретки ←|) аналогична по действию переключателю **E**. Команда **S** ←| приводит к запуску программы с ранее установленного адреса. Следует отметить, что символ ←| на печатающем устройстве не отображается.

Эмулятор пульта управления позволяет приемлемо разрешить проблему управления вычислительной машиной без переключателей. Но работая в рамках даже столь простой операционной системы, невозможно избежать ощущения некоторой отдаленности от машины.

Эмулятор применяется для запуска других операционных систем. На PDP-11/04 используется записанная на гибком диске операционная система **RT-11**. Вызов системы осуществляется командой **DX** ←| в ответ на приглашение программы эмулятора \$, после чего система **RT-11** отвечает набором сообщений и символом **.**, который является символом приглашения ее монитора.

УПРАЖНЕНИЕ. Если по какой-либо причине остановить программу эмулятора, а затем перезапустить, то она распечатает содержимое регистров **R0**, **R4**, **SP** и **PC** перед остановом. Используя эту информацию, придумайте такую последовательность из двух команд, что при ее загрузке в конец ваших программ управление бы передавалось эмулятору по аналогии с обращением к **.EXIT**.

Последовательный вывод литер. На первый взгляд кажется, что программа вывода более чем одной литеры является простым повторением только что описанной. У нас получилось бы тогда что-нибудь вроде

```

                                MOV      #MESSAGE,R1
LOOP:                          TSTB     (R1)
                                BEQ       DONE
                                MOVB      (R1)+,TPB
                                BR        LOOP
                                ...
                                ...
MESSAGE: .ASCIZ                /TESTING OUTPUT/

```

(Чем то, что мы пытаемся сделать, отличается от возможностей монитрного вызова **.PRINT?**) Подобный цикл мы применили бы и для вывода результатов вычислений. Заметьте, что, поскольку мы выводим информацию, хранящуюся в соответствующих байтах массива **MESSAGE**, применение байтовой команды **MOVB** с автоинкрементной адресацией корректно.

Эта простая программа не работает по причине, которая оказывает влияние на все программное обеспечение ввода-вывода: ЦП посылает данные в терминальный буфер вывода гораздо быстрее, чем терминал успевает их обрабатывать. Буфер вывода рассчитан лишь на одну литеру, и, пока текущая литера не будет напечатана, любая попытка заслать туда следующую окажется бесплодной. Итак, поскольку ЦП посылает поток литер слишком быстро, многие из них будут утеряны. Оформите предыдущий фрагмент в виде законченной программы и можете убедиться в этом воочию.

Дабы удостовериться в том, что литера уже напечатана, процессор перед посылкой новой литеры должен сначала проверить, готов ли буфер вывода принять ее. Такая проверка осуществляется обращением к *регистру состояния печатающего устройства*, который обычно называется **TPS**. Так же как и буфер вывода, регистр состояния имеет свой адрес и расположен на одно слово раньше, чем **TPB**. Седьмой бит является битом готовности; он сбрасывается на время занятости буфера вывода и устанавливается в момент, когда буфер готов принять новую литеру. Итак, перед посылкой литеры в **TPB** программа должна ждать, пока седьмой бит **TPS** не будет установлен. Удобно, что седьмой бит есть знаковый разряд младшего байта слова. Наш цикл печати теперь становится таким:

```

                                TPS = 177564
                                TPB = 177566
                                ...
LOOP:    MOV      #MESSAGE,R1
          TSTB    (R1)
          BEQ     DONE
1$:      TSTB     TPS
          SPL     1$
          MOVB    (R1)+,TPB
          BR      LOOP

```

Имейте в виду, что седьмой бит **TPS** устанавливается и сбрасывается электроникой терминального устройства, и только ею. Любая программная попытка изменить его значение игнорируется. Это бит *только для чтения*.

УПРАЖНЕНИЯ. 1. Оформите предыдущий фрагмент в завершенную программу печати текста.

2. Напишите программу печати в десятичном виде содержимого заданной ячейки памяти.

3. Оцените скорость ЦП по отношению к печатающему устройству, включив в цикл с меткой **1\$** счетчик повторений цикла.

Ввод литер. Клавиатура терминала подключена к вычислительной машине так, что можно по адресу получить доступ к буферу клавиатуры и регистру состояния точно так же, как и к буферу вывода. Адреса этих ячеек, как правило, такие:

TKS=177560	; состояние клавиатуры
TKB=177562	; буфер клавиатуры

Набранная на клавиатуре литера преобразуется терминалом в код ASCII и заносится в буфер. Когда терминал работает в автономном режиме, код литеры из буфера клавиатуры пересылается в буфер вывода, и тогда терминал можно использовать как телетайп. Если, однако, терминал подключен к вычислительной машине, то не существует прямой связи между **TKB** и **TPB**. Но даже если она и существует, то может быть отключена. В этом случае терминал закончит обработку вводимой литеры, как только код ASCII литеры попадет в **TKB**. Терминал установит 1 в седьмой бит **TKS**, сообщая тем самым, что литера занесена и доступна в **TKB**. Седьмой бит **TKS** является битом готовности, аналогичным седьмому биту в **TPS**. Литера хранится в буфере только доли секунды. Если за это время ЦП ее не считывает, то она пропадет, а бит готовности будет сброшен, что говорит о готовности буфера принять новую литеру.

Таким образом, действия ЦП должны быть согласованы с работой терминала. Как только литера считывается из **TKB**, терминал очищает бит готовности к **TKS** и способен принять следующую литеру. Напомним, что надо проверять лишь знаковый разряд младшего байта регистра состояния. Поэтому цикл чтения последовательности литер в массив байтов, указанный в **R1**, имеет следующий вид:

LOOP:	TSTB	TKS
	BPL	LOOP
	MOVB	TKB,(R1)+
	BR	LOOP

Поскольку цикл проверки **TKS** будет выполняться много раз, мы можем существенно улучшить программу, используя более подходящий способ адресации:

	MOV	#TKS,R2
LOOP:	TSTB	(R2)
	BPL	LOOP
	MOVB	2(R2),(R1)+
	BR	LOOP

Все же для большей ясности представления материала в приводимых ниже примерах такой подход применяться не будет.

Здесь вся беда в том, что не понятно, как завершить программу. Если мы перестанем набирать литеры, то программа просто-

напросто «зависнет» на двустрочном цикле с меткой **LOOP**. Может быть, сто́ит заканчивать ввод клавишей $\leftarrow|$. Предполагая, что соответствующий код является частью хранимой информации, нужно после записи литеры в массив каждый раз проверять ее на совпадение с **LINE FEED**. Сделать это как будто довольно просто, заменив две последние команды нашего цикла на

```
MOVW      TKB,(R1)
CMPW      #12,(R1)+
BNE       LOOP
```

Однако некоторые системы оперируют с постоянно установленным восьмым битом **TKB**; в других он используется как бит четности. (Вам не нужно беспокоиться о восьмом бите, когда пересылаете данные в коде **ASCII**, поскольку устройство печати терминала его игнорирует.) Поэтому в памяти символ **LINE FEED** будет иметь код **212** с установленным восьмым битом, и сравнение не даст желаемого результата. *Поступить можно так:* перед сравнением с **#12** очистить восьмой бит вводимых данных командой **BICW #200,(R1)**. (Почему нам до сих пор не нужно было при вводе литеры сбрасывать восьмой бит?)

Программа чтения литер с клавиатуры может функционировать неправильно, если она выполняется в то время, когда монитор находится в оперативной памяти. Поэтому, если возможно, поработайте с ней в отсутствие монитора. Чтобы обойтись без монитора, вы можете написать свою собственную программу сбрасывания шестого бита **TKS** перед началом ввода. Смысл этого мы поясним в следующем параграфе. Ваша программа должна также после окончания ввода установить шестой бит.

УПРАЖНЕНИЯ. 1. Напишите программу чтения набираемых на терминале литер, которая заносит их в блок байтов, начиная с адреса **2000**, и заканчивает свою работу по клавише $\leftarrow|$.

2. Загрузите свою программу «вручную», т. е. используя переключатели или эмулятор, и запустите ее. Проверьте, что

- а) набираемые литеры *не* появляются на терминале;
- б) все литеры заносятся в память.

3. Измените программу так, чтобы литеры *высвечивались* на экране терминала. (Получите эхо литер.)

4. Используйте **#** в качестве клавиши удаления, получая при этом эхо стираемой литеры и уничтожая ее из памяти.

Начальные загрузчики. Мы уже говорили о том, что начальная процедура, выполняемая эмулятором пульта, предполагает наличие схем, которые автоматически запускают программу, устанавливающую связь между ЦП и терминалом. Существенная проблема при запуске вычислительной машины состоит в том, чтобы

ввести программу, позволяющую машине осуществлять ввод. Получается замкнутый круг, поскольку любое решение проблемы опирается на предположение о том, что оно уже получено. Обычно здесь проводят параллель с попыткой приподнять себя над землей за волосы. И хотя аналогия вовсе не убедительна, она дала жизнь общепринятому теперь термину¹⁾. Программы, иницирующие связь между процессором и терминалом, называются начальными загрузчиками или предзагрузчиками. На некоторых машинах PDP-11 предзагрузчик нужно вводить вручную (*программный* предзагрузчик), т. е. посредством переключателей. В этом случае руководство оператору должно содержать список заносимых кодов и их адресов. Однако, к счастью, большинство начальных загрузчиков встроено в аппаратуру, и их запуск осуществляется одним переключателем (*аппаратный* предзагрузчик). Эмулятор пульта использует такую процедуру.

Если все же приходится пользоваться программным предзагрузчиком, то он должен быть как можно более коротким, чтобы снизить вероятность ошибок при вводе. Программные предзагрузчики являют собой прекрасные примеры виртуозного программирования и демонстрируют при этом не только выгоду от экономии, но также и отрицательные последствия от неясности.

Обычная функция начального загрузчика состоит в том, чтобы ввести и запустить более длинную, более сложную программу загрузки, которую было бы слишком утомительно заносить вручную. Последняя должна хорошо согласовываться с предзагрузчиком и храниться в некотором постоянном запоминающем устройстве. Таким образом, предзагрузчик должен сначала установить связь с запоминающим устройством, а затем запустить на нем программу, которая в свою очередь установит связь с терминалом и, возможно, будет выполнять самые примитивные функции монитора.

Если такая программа размещена на диске, предзагрузчик должен обращаться к дисковым эквивалентам регистра состояния и регистра данных. Эта процедура слишком сложна, чтобы служить здесь хорошим иллюстративным примером. Если же монитор находится на перфоленте, то он загружается пользователем, и предзагрузчик лишь должен уметь прочитать ее. Хотя в настоящее время перфолента и уступает в популярности электронным носителям информации, ее еще можно встретить в комплектации многих вычислительных машин. Перфолента проходит под считывающей головкой, состоящей из восьми светочувст-

¹⁾ Для процедуры начального запуска в английском языке используется термин *bootstrap*. Буквальный перевод — ремешок на заднике ботинка, при помощи которого его натягивают на ногу. — *Прим. перев.*

вительных элементов (по одному на каждую «дорожку» перфоленты). Каждый элемент способен определить, пробито отверстие в соответствующем месте дорожки на перфоленте или нет. Суммарный эффект, таким образом, состоит в том, что устройство читает последовательность восьмибитовых байтов. Более детально вдаваться в этот механизм нам здесь ни к чему. Буфер устройства чтения с перфоленты **PRB=177552** используется так же, как **ТКВ**. Регистр состояния считывающего устройства **PRS=177550** аналогичен **ТКС**. Однако первый бит регистра состояния есть разряд «готовности к чтению», и всякий раз перед чтением очередного байта его необходимо программно устанавливать. В момент считывания он автоматически сбрасывается, и для чтения следующего байта его нужно установить снова.

Итак, пользователь должен заправить перфоленту с монитором в фотосчитыватель, ввести предзагрузчик с помощью переключателей и запустить вычислительную машину. После загрузки монитор выдаст на терминал сообщение. В руководстве по процессору PDP-11 типа 11/03 имеется текст программы начального загрузчика для машин с максимальным адресом ячейки, которая непосредственно предшествует блоку, отведенному для адресации периферийного оборудования, равным **077776**. Этот текст воспроизведен на рис. 4.1. Перевод на язык ассемблера сделан нами. Обратите внимание на то, что предзагрузчик дол-

077744	016701	START:	MOV	77776,R1
	000026			
077750	012702	S1:	MOV	#352,R2
	000352			
077754	005211		INC	(R1)
077756	105711	LOOP:	TSTB	(R1)
077760	100376		BPL	LOOP
077762	116162		MOVB	2(R1),77400(R2)
	000002			
	077400			
077770	005267		INC	77752
	177756			
077774	000765		BR	S1
077776	177550		.WORD	PRS

Рис. 4.1. Программа начального загрузчика для перфоленты.

жен располагаться с вполне определенного адреса, и в командах ассемблера этот факт учитывается.

Программа начинается с засылки в **R1** адреса **PRS**. В цикле с меткой **LOOP** регистр состояния проверяется уже знакомым нам способом, и в случае поступления байта последний пересылается из **PRB** в память. Заметьте, что перед этим командой

INC (R1) бит готовности в **PRS** был установлен. Затем программа передает управление назад на метку **S1** для ввода следующего байта.

Давайте разберемся в не столь очевидных командах. Меткой **S1** помечена команда, которая пересылает содержимое ячейки **77752** в **R2**. После ее выполнения **R2** содержит **352**. Поэтому первая прочитанная литера будет записана в ячейку с адресом $77400(R2)=77400+352=77752$. Как видите, это то слово, куда первоначально было помещено число **352**, чтобы потом его можно было занести во второй регистр. Обратите также внимание, что следующая команда увеличивает содержимое этой ячейки.

Отсюда ясно, что начальный загрузчик представляет собой самомодифицирующуюся программу. Причину применения такого подхода, однако, невозможно понять, не зная, что программе на перфоленте предшествует *ракорд*, который представляет собой повторение одного и того же байта на протяжении нескольких *дюймов ленты*. Благодаря ракорду снижается риск физического повреждения самой программы и четко обозначается начало программы — это место, где кончается ракорд. В ракорде монитора пробивается код **351**. Как мы уже видели, после считывания байт **351** пересылается в ячейку с адресом **77752**. Содержимое этой ячейки затем увеличивается на единицу, так что при выполнении команды **BR S1** она содержит код **352**, что и было в ней вначале. Таким образом, программа не изменяется при чтении одного байта из ракорда, а значит, любого количества таких байтов.

Любой иной прочитанный с перфоленты байт будет, однако, изменять программу. Рассмотрим, что произойдет при чтении первого, отличного от **351** байта. Когда он считывается, второй регистр содержит код **352**, сохранившийся от предыдущего цикла чтения, и новый байт будет снова занесен в ячейку **77752**. Следовательно, после возвращения на метку **S1** ячейка **77752** содержит новый байт, увеличенный на единицу, и это значение будет загружено затем в **R2**. Второй регистр указывает теперь на новую ячейку, предназначенную для следующего байта. Ее адрес будет начальным адресом программы монитора.

Монитор должен быть загружен в ячейки памяти, которые непосредственно предшествуют программе предзагрузчика. Допустим, что мониторная программа должна быть размещена с адреса **77600**. Это значит, что первый после ракорда байт устанавливает такое значение в **R2**, что команда **MOVB** засылает следующий байт в ячейку **77600**. Как легко видеть, во второй регистр для этого должно быть записано число **200**. Так как текущее содержимое ячейки **77752** перед засылкой в **R2** увеличивается на единицу, то записанный в нее на предыдущем цикле чтения байт должен быть равен **177**.

Итак, после чтения отперфорированной последовательности байтов 351,...,351, 177 ячейка 77752 и регистр R2 содержат код 200.

Заметьте теперь, что программа уже не саомодифицирующаяся! Ячейка 77400 (R2) больше не находится внутри программы, и на каждом цикле чтения увеличение содержимого ячейки 77752 приводит просто к увеличению на единицу значения R2, так что следующий байт заносится в память сразу после предыдущего. Следовательно, при вводе последовательности байтов 351,...,351, 177, *программа* в память, начиная с адреса 77600, загружается программа.

Программа монитора организована так, что последняя ее ячейка находится как раз перед первым словом предзагрузчика. Однако если бы на этом перфолента и заканчивалась, то предзагрузчик продолжал бы ждать дальнейшего ввода: он не передал бы управления монитору. Перфолента должна содержать еще какую-то информацию, которая позволит передать управление на некую ячейку внутри мониторной программы.

С перфоленты будут прочитаны еще восемь байтов. Первые шесть из них совпадают с шестью первыми байтами программы предзагрузчика. При чтении содержимое ячеек 77744, 77746 и 77750 останется таким же, как прежде. Предзагрузчик опять становится саомодифицирующимся, но эти шесть байтов, в сущности, не изменяются. Седьмой байт будет записан в младший байт ячейки 77752. Заметьте, что при входе в цикл чтения этого байта ячейка 77752 опять содержит число 352. Седьмой байт будет равен 373, и потому при выходе на метку S1 по адресу 77752 будет стоять число 374.

Предположим, что управление должно быть передано по адресу 77600, т. е. на первое слово программы монитора. Тогда последний байт должен быть равен 701. Перед входом в цикл чтения этого байта значение R2 равнялось 374, и поэтому этот байт будет занесен по адресу $77400 + 374 = 77774$. Следовательно, команда MOVБ изменит команду BR S1; с новым кодом 000701 это будет BR 77600 (проверьте).

4.2. Прерывания

Если программа ввода литер, написанная для изолированной системы, выполняется в присутствии достаточно совершенного монитора, то могут происходить любопытные вещи. Мы провели эксперимент: запустили под управлением системы RT-11 нашу программу ввода литер, которая не дает эхо, но заносит литеры в память. В результате все литеры *появились* как эхо на терминале, но далеко не все попали в память; более того, при каждом запуске программы в памяти оказывался разный набор

литер. К тому же монитор пытался интерпретировать литеры как команды, что приводило к сообщениям об ошибках.

Причина такого поведения состоит в том, что стандартная функция монитора вошла в противоречие с целями программы. Одной из функций монитора является осуществление ввода с клавиатуры терминала: эхо-печать набираемых литер, приостановка программы при двойном наборе $\wedge C$ и т. д. В нашей программе не было предусмотрено защиты от монитора, который поэтому перехватывал у нее вводимую с клавиатуры информацию. Создавалось впечатление, что во время цикла связи процессора с терминалом программе открывался доступ к тем же самым данным.

В наших предыдущих обсуждениях работы ЦП не было речи о том, каким образом монитор управляет вводом с терминала. Мы говорили, что ЦП выполняет команды последовательно, согласно содержимому регистра РС. Тогда нельзя объяснить, как двумя нажатиями $\wedge C$ можно остановить заиклившуюся программу. Ясно, что при вводе $\wedge C$ монитор обращается к специальной программе. Но как ему удастся перехватить ЦП при чтении первой литеры?

Общая шина. Наш ответ требует некоторого понимания того, как взаимодействуют различные части системы PDP-11. Такая система включает: ЦП, оперативную память, клавиатуру и печатающее устройство терминала, запоминающие устройства, такие, как диски или ленты, и, возможно, некоторые другие периферийные устройства (таймеры, дисплеи и устройства быстрой печати). Было бы невозможно управлять вычислительной системой без электронной линии связи или *шины* между каждым внешним устройством и ЦП. В системе PDP-11 вместо отдельных шин для каждого устройства имеется один общий информационный канал. Он называется *общей шиной* и является стержнем системы PDP-11.

ЦП, память и все внешние устройства связаны с общей шиной. Устройства могут использовать общую шину как для общения с ЦП, так и для непосредственной связи друг с другом, и поэтому те из них, которые могут принимать и передавать данные без помощи ЦП (к ним относятся некоторые типы дисков и магнитных лент), во время этих операций не занимают ЦП. Структура системы PDP-11 изображена на рис. 4.2.

Источник и приемник передаваемых данных распознаются общей шиной по их адресам. Это те же самые адреса, которые использует при работе процессор. Как мы видели, оперативная память занимает несколько тысяч адресов общей шины. Для печатающего устройства терминала (об этом мы уже знаем) доста-

точно всего двух адресов — адреса регистра состояния и адреса регистра данных.

Команды обмена информацией между ЦП и памятью работают потому, что, когда ЦП выполняет их, общая шина находится

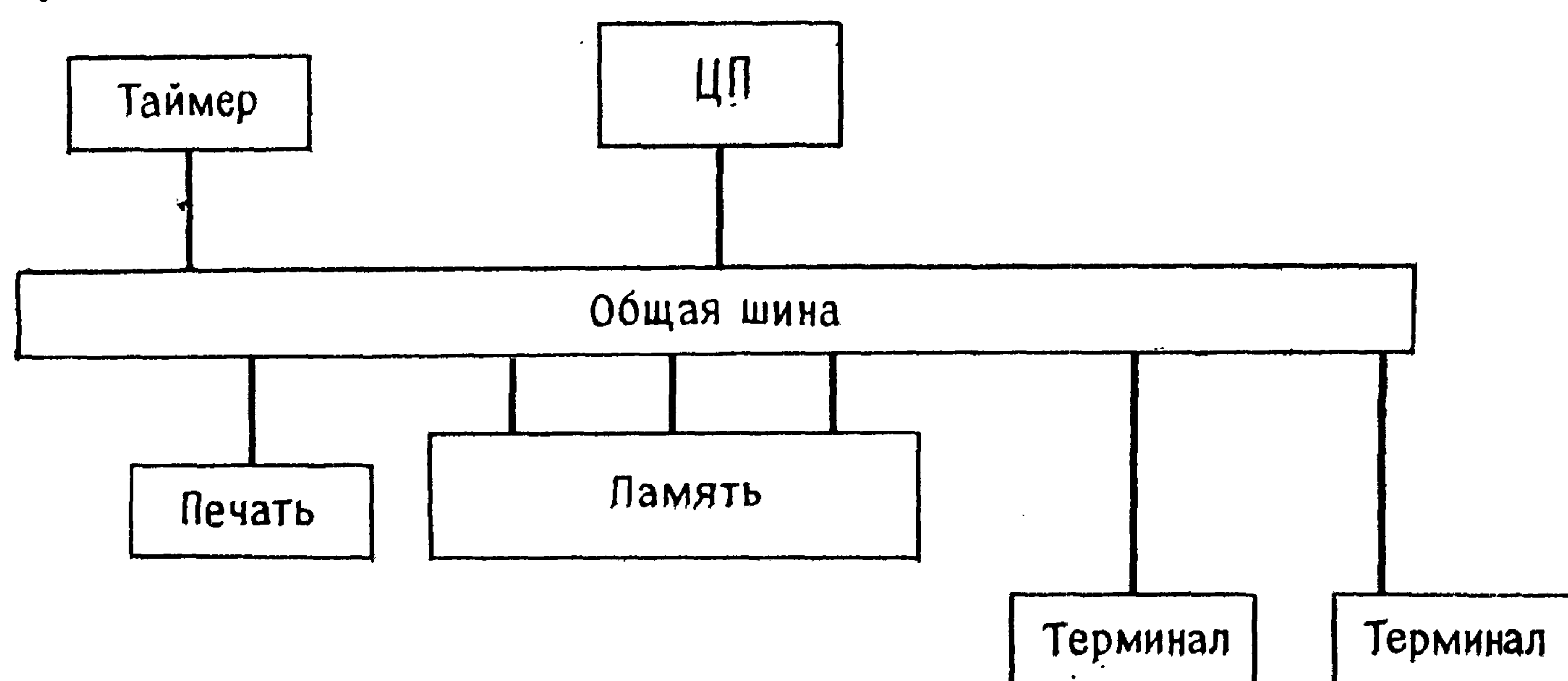


Рис. 4.2. Система PDP-11.

под его управлением. *Только одно устройство в каждый момент времени может управлять общей шиной.* Устройство, получившее управление, может использовать общую шину для передачи или приема данных от любого другого устройства. Понятно, что далеко не все устройства могут распоряжаться общей шиной для такого обмена.

Как правило, общей шиной управляет ЦП. Однако и из других устройств может поступить запрос на управление общей шиной. Например, нажатие клавиши на клавиатуре терминала приводит к посылке терминалом сигнала к ЦП по *специальному каналу запроса*, находящемуся внутри общей шины. Поступление запроса говорит о желании устройства взять на себя управление общей шиной и получить услугу обслуживания центральным процессором. ЦП всегда заканчивает выполнение текущей команды. После этого он проверяет, имеет ли устройство, пославшее запрос (в данном случае клавиатура терминала), достаточный *приоритет* для передачи ему управления. Далее мы остановимся на этом чуть подробнее, а сейчас предположим, что приоритет клавиатуры достаточен и потому ей передается управление общей шиной.

Получив в свое распоряжение общую шину, клавиатура терминала через нее посылает в ЦП *запрос на прерывание*. Этот сигнал вызывает специальный отклик процессора для *обслуживания* запроса. Однако сначала ЦП проверяет, что устройству, пославшему запрос, прерывания *разрешены*. Об этом также будет сказано ниже, а сейчас допустим на время, что клавиатуре разрешены прерывания. Запрос на прерывание, посылаемый устройством процессору, сопровождается адресом *вектора прерывания* (или, что то же самое, адресом ячейки памяти), т. е. вместе с

запросом на прерывание клавиатура передает такую информацию: «ячейка 60». Выбор именно такого адреса обусловлен конструкцией устройства сопряжения или *интерфейсом* между терминалом и общей шиной. ЦП посылает теперь сигнал подтверждения, после которого управление общей шиной возвращается ЦП.

И вот теперь следует отклик ЦП. Он начинается с сохранения слова состояния и счетчика команд в системном стеке. Заметьте, что для ссылки на слово состояния процессора **PS** служит адрес ячейки **177776** (о возможных вариациях для восемнадцати и двадцатидвухразрядных адресов упоминалось в предыдущем параграфе). В результате такого действия состояние процессора в момент прерывания сохраняется. Затем, трактуя переданный адрес как адрес первого из двух слов, ЦП загружает в **PC** содержимое первого слова, а в **PS** содержимое второго. Важно подчеркнуть, что весь отклик, включая выбор адресов, откуда поступают новые значения **PC** и **PS**, выполняется *аппаратурой*.

Для терминальной клавиатуры отклик процессора эквивалентен следующей последовательности команд:

MOV	PS, -(SP)
MOV	PC, -(SP)
MOV	60, PC
MOV	62, PS

Итак, аппаратный отклик на прерывание завершен, и ЦП возвращается к своему обычному делу — к исполнению команд. Но поскольку **PC** изменился, будут выполняться не те команды, которые выполнялись бы, не случись прерывания. Ячейка **60** должна быть установлена предварительно либо монитором, если таковой имеется, либо самим пользователем и должна указывать на подпрограмму ввода литеры. Эта подпрограмма называется *подпрограммой обработки прерываний* от клавиатуры. После приема литеры она может возвратить управление главной программе, но это не обязательно. Ячейка **60** может указывать на любую подпрограмму пользователя, которую тот напишет. Поскольку процедура обработки прерывания пользуется аппаратным стеком, под него должен быть зарезервирован блок памяти, максимальный адрес которого находится в регистре **SP**. Не забывайте предусматривать это в тех программах, которые будут выполняться без монитора.

Разрешение прерываний. Мы упомянули, что перед обслуживанием прерывания ЦП проверяет, имеет ли запрашивающее устройство *разрешение* на прерывания. Если они ему не разрешены, то ЦП игнорирует запрос.

В автономной системе предоставление разрешения на прерывания находится в руках программиста. В случае клавиатуры

терминала оно управляется шестым битом **TKS**. Если он установлен, то прерывания разрешены, в противном случае — нет.

При включении вычислительной машины шестой бит сброшен, но он устанавливается во время генерации операционной системы. Под управлением системы **RT-11** мы выполнили следующую программу, которую вам следует попытаться пропустить самостоятельно:

```

                                .TITLE      TKOFF
                                TKS = 177560
START:                         BIC          #100,TKS
1$:                             BR          1$
                                .END         START

```

Эта программа делает невозможными прерывания от клавиатуры терминала. В результате клавиатура становится «мертвой», и программа не может быть остановлена двумя командами **^C**.

Аналогичная программа, в которой вместо вхождения в цикл осуществляется выход, не сможет защитить от прерывания с клавиатуры. Это происходит потому, что в системе **RT-11** мониторный вызов **.EXIT** каждый раз восстанавливает разрешение на прерывания, препятствуя именно той цели, ради которой программа была написана.

УПРАЖНЕНИЯ. 1. Напишите программу выдачи какого-нибудь сообщения, которая запрещает прерывания от клавиатуры в процессе вывода и снова разрешает их после его окончания.

2. Шестой бит **TPS** отвечает за разрешение прерываний от печатающего устройства. Что вы понимаете под запросом на прерывание от печатающего устройства? Что произойдет, когда вы установите шестой бит **TPS**? Почему?

Поступление прерывания не влияет на шестой бит **TKS** или **TPS**. Один раз установленный, он сохраняется до тех пор, пока эта или другая программа не сбросит его или пока система не будет остановлена и заново запущена. Поскольку мониторы для ввода используют прерывания от терминальной клавиатуры, они никогда не запрещают прерывания. Поэтому в дальнейшем мы будем считать, что прерывания от клавиатуры всегда разрешены.

Для иллюстрации прерывания от клавиатуры рассмотрим фрагмент, который вызывает выход в систему при нажатии любой клавиши:

```

START:                         MOV          60,R0
                                MOV          #SERV,60
1$:                             BR          1$
SERV:                         MOV          R0,60
                                .EXIT

```


Монитор позволяет изменять содержимое ячеек прерывания. Поэтому в приведенном фрагменте перед засылкой в ячейку 60 адреса нашего собственного фрагмента обработки прерывания и входом в бесконечный цикл сохраняется содержимое этой ячейки в **R0**. Любое прерывание от клавиатуры приведет к занесению в счетчик команд **PC** адреса метки **SERV**, в результате чего наша подпрограмма восстановит первоначальное значение ячейки 60, чтобы остался путь в системный фрагмент обработки прерывания, и осуществит выход.

Единственная загвоздка здесь состоит в том, что в момент выхода нашей подпрограммы содержимое **PS** по-прежнему совпадает с содержимым ячейки 62. Так уж получилось, что в системе, в которой мы работали, решение любых проблем такого рода берет на себя мониторный вызов **.EXIT**. Если же, как чаще всего случается, вместо выхода нам нужно после обработки прерывания вернуться в главную программу, мы должны воспользоваться командой **RTI**. Как вы помните, эта команда значения **PC** и **PS** возьмет из стека, после чего состояние ЦП становится в точности таким, каким оно было до прерывания. Конечно, так же как и в команде **TRAP**, адрес возврата может быть изменен: для этого достаточно изменить значение (**SP**).

УПРАЖНЕНИЯ. 1. Измените предыдущий фрагмент так, чтобы при вводе любой литеры, кроме буквы **X**, происходил возврат в цикл, а при вводе **X** осуществлялся выход.

2. Что произойдет, если программа обработки прерываний зациклится?

Мы использовали механизм прерываний, чтобы прекратить выполнение программы, но, конечно, он может быть применен и для ввода данных. Давайте построим программу, которая делает два дела «сразу»: из одного блока памяти посылает данные печатающему устройству и в то же время пересылает вводимые с клавиатуры данные в другой блок памяти.

Главная программа сохраняет содержимое ячейки 60 в регистре, заносит в эту ячейку адрес программы обработки прерываний **SERV** и выводит сообщение, первый байт которого указан в другом регистре. Подобный фрагмент вывода встречался в § 4.1. После этого программа восстанавливает содержимое ячейки 60 и осуществляет выход.

Тем временем программа обработки прерываний может заносить вводимые данные в блок байтов, на который указывает регистр **R2**:

```
SERV:      MOVB      TKB,(R2)+
           RTI
```


Заметьте, что программа обработки прерываний не требует наличия цикла проверки бита готовности в **TKS**. Действительно, поскольку клавиатура вызвала прерывание, очередная литера должна быть доступна. Это приводит к значительной экономии процессорного времени и к тому, что ввод-вывод посредством механизма прерываний оказывается существенно более эффективным (*управляемый по прерываниям ввод-вывод*).

УПРАЖНЕНИЯ. 1. Объедините описанные выше главную и обрабатывающую прерывания подпрограммы в законченную программу. Нарисуйте ее блок-схему.

2. Измените программу так, чтобы она прекращала свою работу после завершения ввода и вывода и не зависела от того, что закончится раньше. Не потребуются ли новые, отсутствующие в программе, действия по координации?

3. Напишите свое собственное макроопределение **.TTYIN**.

4. Напишите управляемую по прерываниям процедуру для ввода десятичного числа с клавиатуры терминала.

5*. Напишите программу, которая упорядочивает числа по возрастанию по мере их ввода и с этой целью просматривает блок, где уже хранятся введенные и упорядоченные числа, находит подходящее место для нового числа, заносит его туда и сдвигает на одну ячейку остальную часть блока. Включите в нее управляемый по прерываниям фрагмент для чтения чисел. Убедитесь, что вы не делали никаких предположений об относительных скоростях протекания различных процессов. Какой метод сортировки в сравнении с приведенным в § 2.3 более эффективен? Какая программа более эффективна?

6. Добавьте к вашей программе из упр. 5 фрагмент, который позволил бы вам оценить процессорное время (в единицах длительности некоторого цикла команды), которое теряется каждым процессом на ожидание завершения другого.

Прерывания для печатающего устройства. Мы убедились в том, что, имея прерывание от клавиатуры терминала в момент ввода литеры, можно высвободить ЦП для другой работы и не растрачивать его возможности на холостые циклы в ожидании очередной литеры. Естественно, что для получения реального выигрыша программы должны быть устроены так, чтобы процессор был занят решением другой задачи, а не простаивал в каком-то еще цикле в ожидании следующего прерывания.

Точно так же намного предпочтительней иметь управляемый по прерываниям вывод и не держать процессор в цикле опроса регистра состояния печатающего устройства до тех пор, пока не освободится буфер. Условием, вызывающим прерывание, в данном случае будет отсутствие литеры в **TRV** (а точнее, установка

бита готовности в **TPB**). Конечно, было бы неудобно, если бы отсутствие литеры в **TPB** *всегда* вызывало прерывание. Представьте себе последствия такого решения в программе, в которой нет или почти нет операций вывода! Ясно, что нужно запретить прерывания от устройства печати, за исключением тех моментов, когда в программе действительно есть данные для вывода.

Пробная программа, демонстрирующая эффекты от разрешения прерываний печатающему устройству командой **BIS #100, TPS**, окажется бесполезной, если вы запустите ее с монитором. Через ячейку прерывания для печатающего устройства управление будет передано мониторной программе обработки прерываний. Обнаружив, что печатать нечего, эта программа, выполняя одну из своих функций, *запретит прерывания от устройства печати* и возвратится в вашу программу.

УПРАЖНЕНИЕ. Какую информацию вы могли бы получить, пропустив такую программу без монитора?

Адрес вектора прерывания печатающего устройства равен **64**. Давайте снова — и не в последний раз — напомним программу печати буквы **B** на терминале. Она приведена на рис. 4.3. Обра-

```

                .TITLE  BPRINT
                R1=%1
                TPS=177564
                TPB=177566
START:         MOV     64,R1
                MOV     #SERV,64
                BIS     #100,TPS
                HALT
SERV:          MOVB    #102,TPB
                BIC     #100,TPS
                MOV     R1,64
                RTI
                .END    START

```

Рис. 4.3. Программа для иллюстрации механизма прерываний от печатающего устройства.

тите внимание на то, что программа восстанавливает **PS** так же, как и в случае прерываний от клавиатуры, и что на выходе она запрещает прерывания от устройства печати.

УПРАЖНЕНИЯ. 1. Напишите программу печати произвольного сообщения.

2. Напишите программу печати содержимого заданной ячейки в десятичном виде.

3. Напишите свое собственное макроопределение **.TTYOUT**.

Если программа предназначена для выполнения управляемого по прерываниям вывода, то разумно на протяжении всего вывода оставлять адрес подпрограммы обработки прерываний в ячейке 64, восстанавливая его при полном окончании вывода. Может показаться, что, поскольку прерывания запрещены, когда нет данных для вывода, никакого вреда не будет, если также поступать даже в присутствии монитора. Посмотрим, однако, что случится, если нажать клавишу терминала. Процессор откликнется на прерывание от клавиатуры переходом через ячейку 60 на мониторную программу обработки прерываний. Одна из ее частей посылает эхо вводимой литеры на устройство вывода. Теперь уже механизм вывода по прерываниям использует монитор. Поэтому введенная литера заносится в такую ячейку, из которой ее может взять *мониторная программа* обработки прерываний печати, и разрешает прерывания от печатающего устройства. Теперь ЦП через ячейку 64 осуществляет переход по установленному пользователем адресу, который указывает на его программу обработки прерываний. В результате выполняется программа пользователя. Но любые изменения содержимого регистров, внесенные мониторной программой обработки прерываний от клавиатуры, могут неблагоприятно сказаться на работе программы пользователя. Как правило, мониторные подпрограммы перед выходом восстанавливают содержимое регистров (кроме R0), но программа, о которой шла речь, была прервана прежде, чем получила возможность сделать это. Когда программа пользователя будет завершена, командой RTI управление будет передано на следующий адрес *мониторной программы обработки прерываний от клавиатуры*. Результат совершенно непредсказуем.

Системные мониторы настолько сложны, что довольно рискованно переносить в программу пользователя часть какой-либо из их функций. Даже при самом детальном изучении монитора можно не заметить какую-то взаимосвязь оставленной и заменяемой частей. Что касается ввода-вывода по прерываниям, то программа, обеспечивающая вывод, должна либо иметь свою собственную подпрограмму обработки прерываний от клавиатуры, либо запрещать прерывания от клавиатуры.

УПРАЖНЕНИЯ. 1*. Придумайте иное решение только что затронутого вопроса, включив в программу обработки прерываний от печати проверку того, является ли вызвавшая прерывание программа программой пользователя. Если нет, то программа пользователя должна уступить управление мониторной процедуре.

2. Напишите программу, в которой управляемый по прерываниям ввод-вывод применяется для вывода эхо набираемых на терминале литер. В ней вам надо зарезервировать два

блока для хранения данных (два буфера). Начните с заполнения первого буфера вводимыми данными, потом заполните второй буфер, затем снова заполните буфер 1 и т. д. Между тем, как только первый буфер будет заполнен, выдайте его содержимое на печать, затем напечатайте данные из буфера 2, потом снова перейдите к буферу 1 и т. д. Перед распечаткой буфера программа должна ждать, пока он не будет заполнен, и наоборот. В результате вводимые литеры будут отображаться на терминале вовремя. Это пример *ввода-вывода с двойной буферизацией*. (Совет: используйте сопрограммы.)

Приоритет. Ранее мы упоминали, что ЦП уступает управление общей шиной только устройству, имеющему достаточный приоритет. Поэтому устройство, приоритет которого слишком низок, не будет даже иметь возможности послать запрос на прерывание.

Любое внешнее устройство, способное вызывать прерывания, имеет свой собственный фиксированный приоритет в диапазоне от 0 до 7. Реально используются только приоритеты 4, 5, 6 и 7. *Уровень приоритета определяется аппаратурой.* Так, обычно приоритет клавиатуры терминала и устройства печати равен 4, и если он должен быть изменен, то необходима физическая переналадка. (На машине LSI-11 только два уровня приоритетов 0 и 1, и все внешние устройства имеют приоритет 1.)

Кроме того, ЦП имеет свой собственный приоритет в диапазоне от 0 до 7 (от 0 до 1 для машины LSI-11). *Приоритет ЦП устанавливается программно.* В каждый момент времени ЦП будет уступать управление общей шиной только устройству с более высоким, чем у ЦП, приоритетом.

Процессор определяет свой текущий приоритет по значению битов 5, 6 и 7 слова состояния **PS** (на LSI-11 только по биту 7). Учтите, что эти три бита относятся к разным цифрам восьмеричной формы представления **PS**.

Покажем, как установить седьмой приоритет процессора, при котором ни одно внешнее устройство не сможет осуществить прерывание. В машинах PDP-11/45 и 11/55 для привилегированных пользователей существует команда установки уровня приоритета **SPL** (Set Priority Level):

SPL

7

В машине PDP-11/45, которой мы пользуемся, эта команда независимо от устанавливаемого приоритета запрещает *любые* прерывания до тех пор, пока не будет выполнена следующая за **SPL** команда. В руководствах этот факт не нашел отражения, и поэтому не ясно, так же ли будут вести себя другие модели.

Если в машине отсутствует команда **SPL**, то можно выполнить команду

MOVB #340,PS ; уровень приоритета 7

поскольку, как мы помним, такая команда сбрасывает разряды условий. Можно было бы обойтись командой **MOV**, но, так как в более мощных моделях PDP-11 используется старший байт **PS**, лучше его не трогать без нужды. Добавим, что для установки именно такого уровня приоритета можно применять также команду **BIS**.

На процессорах LSI-11 к слову состояния прямо адресоваться нельзя, но есть специальные команды «чтение из **PS**» (Move From **PS**) и «запись в **PS**» (Move To **PS**), которым доступен младший байт **PS**:

MTPS #300 ; установка разряда приоритета

УПРАЖНЕНИЯ. 1. Напишите программу печати приоритета процессора, при котором выполняется эта программа.

2. Напишите программу, которая инициирует ввод числа (в диапазоне от 0 до 7) и затем устанавливает соответствующий приоритет.

3. Напишите программу, которая позволит вам узнать приоритет прерывания от клавиатуры терминала. (*Совет: не используйте управляемый по прерываниям вывод.*)

4*. Напишите программу, которая позволит вам узнать приоритет прерывания от печатающего устройства.

Проницательный читатель уже, наверно, догадался, что приоритет ЦП автоматически сбрасывается, когда происходит прерывание. Содержимое **PS** до прерывания заносится в аппаратный стек и замещается содержимым второго слова вектора прерывания. Последнее задается программно. Следовательно, есть два пути для установки приоритета программы обработки прерывания. Допустим, к примеру, что нам желательно, чтобы обрабатывающая прерывания от клавиатуры программа взаимодействовала с ЦП, имея приоритет 4. Тогда у нас есть выбор между

MOVB #200,PS

в самой программе обработки прерывания и

MOVB #200,62

до момента прерывания. Последний способ имеет, конечно, то преимущество, что установка приоритета выполняется только од-

нажды, а не всякий раз, когда происходит прерывание. Однако еще более важное преимущество предварительной установки приоритета через вектор прерывания состоит в том, что, когда оно происходит, аппаратура полностью отреагирует на него прежде, чем допустит другие прерывания. Так, если содержимое ячейки 62 равно 340, любая обрабатывающая прерывания от клавиатуры программа будет иметь приоритет 7, и потому ей не угрожает опасность оказаться прерванной. Однако, если **PS** устанавливается внутри программы обработки прерывания пусть даже ее первой командой вполне возможно вмешательство другого прерывания.

Нетрудно понять, почему иногда нежелательно допускать прерывание обрабатывающей прерывания программы. Например, программа обработки прерываний от клавиатуры может быть прервана до того, как она успела принять литеру из **ТКВ**, а к моменту возврата из этого второго прерывания содержимое **ТКВ** может быть нарушено. Поэтому правильно поддерживать высокий приоритет ЦП до тех пор, пока программа обработки прерываний не выполнит все такие критические действия. Соответственно, если какому-либо устройству прерывания разрешены в любой момент, обрабатывающая эти прерывания программа должна быть короткой. Кроме того, поскольку заранее не известно, в каком месте произойдет прерывание, до команды **RTI** нельзя изменять хранящийся в стеке адрес возврата.

Если эти предосторожности приняты во внимание, то нет проблемы *вложения* прерываний, подобно подпрограммам или программным прерываниям. Более того, возможна любая степень вложения и подпрограмм, и программных прерываний, и прерываний с подходящим использованием команд **RTS** и **RTI**, обеспечивающим корректные выходы.

УПРАЖНЕНИЯ. 1. Может ли программа обработки прерываний от некоторого устройства быть прервана самим устройством?

2. Переделайте все ваши программы с управляемым по прерываниям вводом-выводом, добавив в них подходящую расстановку приоритетов.

Если ЦП получает одновременно запросы на прерывание от нескольких устройств с различными приоритетами, то сначала удовлетворяется запрос, имеющий больший приоритет. Между устройствами с одинаковым приоритетом определяющим фактором является близость к ЦП по общей шине. Сказанное, конечно, относится к приоритету самого устройства, как он определяется аппаратурой, но здесь не подразумевается программно устанавли-

ливаемый приоритет процессора при выполнении обрабатываемой прерывание программы.

Устройство, которому не удалось прервать, не будет забыто. Его запрос будет обслужен, как только позволит уровень приоритета ЦП. А до тех пор прерывание будет *висящим*. Обычно программа обработки более приоритетного устройства будет выполняться ЦП с высоким приоритетом. Менее приоритетный запрос будет висеть до тех пор, пока обслуживающая другое устройство программа не закончит свои действия и не осуществит возврат.

Одним из самых высокоприоритетных устройств, которое, как правило, имеется в системе, является таймер. «Тиканье» таймера есть не что иное, как запрос на прерывание. Частота таких запросов определяется электрической сетью: 60 раз в секунду в США и 50 — в большинстве других стран. Для обработки прерываний от таймера используются ячейки 100 и 102. Регистр состояния таймера имеет адрес 177546; шестой бит регистра, как обычно, служит для разрешения или запрещения прерываний.

Программу обработки прерываний от таймера можно использовать для слежения за течением времени путем подсчета числа «тиканий». Заметьте, что, если запрос на прерывание от таймера не будет обработан в течение $1/60$ с, следующий запрос уже невозможно будет отличить от предыдущего, поскольку процессор может определить только наличие или отсутствие запроса. Поэтому отмеряющая время программа должна внимательно следить за тем, чтобы приоритет ЦП слишком долго не оставался более высоким или равным приоритету таймера, дабы не потерять единицу времени.

УПРАЖНЕНИЯ. 1. Почему таймер не имеет регистра данных?

2. Напишите программу для включения звонка ($\wedge G$) через одnoseкундные интервалы. Оценивали ли вы время работы вашей подпрограммы обработки прерываний от таймера? Оправдались ли эти оценки? (Вам может пригодиться команда **WAIT**; она приостанавливает процессор до поступления очередного прерывания, причем **PC** указывает на следующую команду.)

3. Напишите программу, позволяющую вам определять приоритет таймера.

4. Напишите фрагмент программы, с помощью которого можно было бы перед выходом напечатать время работы программы.

Бит Т. Невозможно до конца понять взаимодействие между трассировочными прерываниями и внешними прерываниями, не разобравшись досконально в аппаратуре. Тема эта довольно

сложная, и здесь нам не остается ничего иного, как указать на несколько подводных камней.

Будем считать, что бит **T** установлен, если занесена 1 в четвертый бит слова **2(SP)** непосредственно перед выполнением **RTI**. Дело в том, что на большинстве моделей PDP-11 бит **T** можно установить только косвенно командами типа **RTI** или **TRAP**. Попытка вроде

BIS

#20,PS

просто не срабатывает, также как и нажатие пульта переключателя **D**.

Нормальная последовательность действий состоит в том, что ЦП выполняет одну команду, следующую за командой установки бита **T**, и затем возникает программное прерывание. Следовательно, если **RTI** устанавливает бит **T**, то одна команда в программе, которой **RTI** возвращает управление, будет исполнена до возникновения трассировочного прерывания¹⁾. Причем, если исполняемая команда есть **SPL**, прерывание откладывается до тех пор, пока процессор не выполнит команду, следующую за **SPL**.

Проблемы возникают, когда программа (предположим **ODT**), устанавливающая бит **T**, выполняется процессором с высоким приоритетом, а та, которой **RTI** возвращает управление, — с низким. Допустим, что во время исполнения команды **RTI** висело прерывание и что оно имеет более высокий приоритет, чем программа, в которую произошел возврат. До того как выбрать команду программы, выполняемой в пошаговом режиме, процессор откликается на прерывание и загружает новое содержимое в **PS** и **PC**. В результате бит **T** оказывается сброшенным, и программа обработки прерывания будет выполняться, не вызывая трассировочных прерываний, и осуществит возврат через свою собственную команду **RTI**. По техническим причинам последняя «немедленно обратит внимание процессора» на то, что бит **T** установлен. Следовательно, трассировочное прерывание возникнет до исполнения первой команды в пошаговом режиме.

Этого можно избежать, заканчивая программу обработки прерываний командой **RTT**²⁾ вместо **RTI**. Единственная разница между ними заключается в том, что **RTT** откладывает трассировочное прерывание на одну команду. Поэтому требуемая команда

¹⁾ Это утверждение и последующее изложение автора противоречат описанию процессоров PDP-11/04, 34, 45, 55. См. "PDP-11/04/34, 45/55 Processor handbook". — Прим. перев.

²⁾ Не реализована на некоторых малых моделях.

будет исполнена, после чего ЦП «заметит» бит Т и возбудит трассировочное прерывание.

Допустим, однако, что висящего прерывания нет, но возникло новое прерывание после того, как выбрана команда, выполняемая в пошаговом режиме. ЦП завершит ее, выполнит программу обработки прерывания и возвратится назад. На этот раз использование для возврата команды **RTT** приведет к тому, что трассировочное прерывание задержится до выполнения еще одной команды; отладчик будет считать, что выполнена лишь одна команда, тогда как на самом деле выполнятся две. Здесь более правильно осуществлять возврат командой **RTI**.

К сожалению, сама природа прерываний допускает существование любого из этих случаев, так что не видно корректного пути выполнения выхода из программы обработки прерываний. Это приводит к неприятностям при отладке работающих по прерываниям программ.

4.3. Внешние запоминающие устройства под управлением монитора

До сих пор вся необходимая нашим программам информация вводилась с терминала в процессе исполнения, а результаты вычислений выводились на терминал. В этом и следующих параграфах мы рассмотрим, каким способом программа может получать данные и записывать их в файл на запоминающем устройстве. Подход здесь общий, *не зависящий* от того, является ли запоминающее устройство каким-либо из многочисленных типов: дисков или лент. Основное внимание в данном параграфе уделяется вводу-выводу под управлением монитора. Краткое описание трудностей, встречающихся при управлении вводом-выводом самим пользователем, приведено в § 4.4.

Мониторные вызовы. Будем описывать ввод-вывод под управлением монитора в терминах макрокоманд системы RT-11. Так же как и в § 1.4, вам необходимо выяснить, имеются ли эти средства в вашей системе, и если нет, то что заменяет их.

В процессе развития операционной системы RT-11 было несколько версий. К моменту написания этой книги самой новой была версия 3В. Большинство системных макрокоманд, рассматриваемых в этом параграфе (но не те, которые обсуждались ранее), различаются в последовательности вызовов и генерируемых кодах в зависимости от используемой версии системы RT-11. Мы будем изучать только самую последнюю версию этих макрокоманд. Если системная макробibliothek вашей системы RT-11 относится к версии 3 или 3В, вы можете использовать эти макро без исправлений, если по поводу каждой из них будет упомина-

ние в директиве **.MCALL**, которая указывает ассемблеру, чтобы он произвел поиск в системной макробιβлиотеке **SYSMAC.SML**. Где-то в начале распечатки этого файла сообщается о номере версии.

Как упоминалось в § 1.4, ассемблер и системная макробιβлиотека должны быть совместимы. В нашем распоряжении была система PDP-11, в которой имелись все версии ассемблера MACRO-11 системы RT-11 и только версия 2 бιβлиотеки **SYSMAC.SML**. К сожалению, формат бιβлиотечных файлов версии 2 таков, что они не могут быть прочитаны третьей версией ассемблера, и поэтому, чтобы использовать системные макро, мы должны либо писать собственные последовательности вызовов для мониторных программ **ЕМТ**, либо применять вторую версию ассемблера.

Системные макро, о которых пойдет речь, не будут работать правильно, если они без каких-либо изменений транслируются более ранними версиями ассемблера. Из-за различий в способах передачи параметров возникнут еще и ошибки при трансляции. Так, в вызовах третьей версии передается адрес числа, которое попадет в младший байт команды **ЕМТ**, а в ассемблерах второй версии предполагается, что передается само число; поэтому в младшем байте команды **ЕМТ** окажется адрес. (Почему это вызовет ошибку при *трансляции*?)

Вы можете использовать макровывзов третьей версии с ассемблером и бιβлиотекой версии 2, если включите в вашу программу системный макровывзов **..V2..**, выполняющий преобразование в формат второй версии. Имейте в виду, что имя этой макрокоманды содержит *шесть* литер. Таким образом, в вашей программе должны быть строки

```
                .MCALL    ..V2..  
                ...  
                ...  
START:         ..V2..
```

и тогда мониторные вызовы третьей версии будут корректно интерпретироваться операционной системой второй версии. Если в вашем распоряжении система первой версии, замените **..V2..** на макрокоманду **..V1..**. (Если вы включите в программу обе макрокоманды, чтобы она работала в обеих системах, она не будет работать ни в одной из них.)

УПРАЖНЕНИЕ. Выясните, как работает **..V2..**.

50-ричный код. С нашей точки зрения, преимущество организованного через монитор взаимодействия между программой и запоминающим устройством заключается в возможности обраче-

Литера	Код	
Пробел	0	ния к файлам по именам способом, ко-
A	1	торый нам уже знаком. В программе
.	.	же, которая сама осуществляет ввод-вы-
.	.	вод, приходится выполнять массу спе-
Z	32	цифических действий, чтобы определить
\$	33	конкретное физическое место, где на-
Точка	34	ходятся ее данные.
Не используется	(35)	Поэтому программа должна уметь
0	36	передавать имена файлов монитору. В
1	37	PDP-11 системные программы кодируют
2	40	такие имена не в коде ASCII, а в 50-
.	.	ричном коде. Это такой код, в котором
.	.	могут быть представлены только заг-
9	47	лавные буквы, цифры, . (точка), про-
		бел и \$ (см. слева).

Таким образом, имена файлов состоят из символов, которые кодируются числами, меньшими 50 (восьмеричное).

Мы можем рассматривать эти символы как цифры в системе счисления с основанием 50 (восьмеричное). В такой системе запись XY нужно интерпретировать так: X — в столбце «50-ок», Y — в столбце единиц. Поскольку в рассматриваемой кодировке X представляется числом 30, а Y — числом 31, то XY представляется числом: $(30 \times 50) + 31 = 1731$ (вычисления в восьмеричной системе). Аналогично YX есть $(31 \times 50) + 30 = 1750 + 30 = 2000$.

Подобным же образом «трехразрядное» 50-ричное число XYZ представляется восьмеричным числом:

$$(30 \times 50 \times 50) + (31 \times 50) + 32 = 113000 + 1750 + 32 = 115002$$

Заметим, что такое представление трех символов XYZ кодируется в одно слово машины PDP-11. Действительно, наибольшее состоящее из трех «цифр» число в 50-ричной системе есть 999, которое в восьмеричном виде записывается так:

$$(47 \times 50 \times 50) + (47 \times 50) + 47 = 171700 + 3030 + 47 = 174777$$

Оно также может быть закодировано в одно слово машины PDP-11. Итак, применение 50-ричного кода позволяет нам кодировать три литеры в одно 16-разрядное слово, если только литеры берутся из специального набора.

УПРАЖНЕНИЕ. Напишите программу, которая вводит три литеры и печатает соответствующий им 50-ричный код.

Директива **.RAD50** используется для перевода цепочки литер в 50-ричный код с упаковкой по три литеры в слово. Синтаксис

этой директивы совпадает с синтаксисом директивы **.ASCII**. Если для последнего слова останутся только одна или две литеры, то к ним справа будут добавлены пробелы (пробел, как вы помните, имеет код 0). Ниже приведен пример кодировки (символом **#** обозначен пробел):

114750	.RAD50	/XY/	;эквивалентно/XY#/
001731	.RAD50	/#XY/	

Драйверы устройств. Для каждого устройства, с которым монитор способен взаимодействовать, имеется специальная программа, входящая в состав операционной системы и называемая *драйвером* устройства. Драйвер столь часто используемого устройства, как диск, будет, по-видимому, постоянно в резидентной части монитора. Драйверы же других устройств могут находиться на диске и загружаться в память лишь по мере необходимости.

Прежде чем получить доступ к устройству, его драйвер нужно загрузить в память. Если операционная система этого не сделала, программа должна сама произвести загрузку, применив мониторный вызов **.FETCH**. До тех пор, пока вы досконально не изучите вашу операционную систему, нужно каждый раз применять вызов **.FETCH**. Даже если драйвер уже находится в памяти, никакого вреда от этого не будет.

Мониторный вызов **.FETCH** имеет два отделяемых запятыми параметра: адрес, по которому должен быть загружен драйвер, и адрес, по которому записано имя устройства. Таким образом, директива

.FETCH #HNDLER,#DEVNAM

заставляет монитор взять из ячейки **DEVNAM** имя устройства и загрузить соответствующий ему драйвер в память, начиная с ячейки **HNDLER**. Обратите внимание на синтаксис этой директивы: в обоих параметрах используется непосредственная адресация.

В ячейке **DEVNAM** должно находиться двухбуквенное имя устройства в 50-ричном коде. Примеры таких кодов: диск **RP04** — **DK**; диск **RK06** — **DM**; лента **DEC** — **DT**; кассетная лента — **CT**; устройство чтения с перфокарт — **CR**; перфолента — **PC**. Взяв в качестве примера диск **RP04**, будем иметь

DEVNAM: .RAD50 /DK/

Заранее не известно, сколько места займет драйвер. Поэтому разумно метку **HNDLER** поставить непосредственно перед директивой **.END**, чтобы драйвер устройства был загружен в свободную область памяти. Монитор в ответ на вызов **.FETCH**

занесет в нулевой регистр адрес первого следующего за драйвером слова. Если же драйвер уже находился в памяти, то монитор лишь занесет в R0 адрес метки **HNDLER** и не будет загружать копию драйвера.

Таким образом, схема следующая:

```

                .MCALL      .FETCH
START:          ...
                ...
                ...
                .FETCH      #HNDLER,#DEVNAM
DEVNAM:         .RAD50      /DK/
                ...
HNDLER:         ...
                .END        START

```

Заметьте, что под программу драйвера здесь на самом деле места не отводится. (Как можно это сделать? Почему нам это может понадобиться?)

Можно загружать драйверы нескольких устройств:

```

                .FETCH      #HNDLER,#DEV1
                .FETCH      R0,#DEV2
                .FETCH      R0,#DEV3

```

и т. д. Обратите внимание на синтаксис: при первом вызове передается адрес ячейки **HNDLER** и поэтому применяется непосредственная адресация; в дальнейшем же *содержимое* нулевого регистра указывает адрес, с которого начинается загрузка драйвера.

Макрокоманды системы RT-11 сбрасывают бит С в случае успешного завершения своих функций и устанавливают его, если обнаруживается ошибка. Операции ввода-вывода с запоминающими устройствами могут оканчиваться неудачей по разным причинам, многие из которых (например, диск выключен) находятся вне программного контроля. Поэтому необходимо после каждого мониторингового вызова анализировать бит С командой **BSC** и передавать управление фрагменту, который печатает сообщение об ошибке и затем либо воспринимает указание с терминала о дальнейших действиях, либо просто осуществляет выход в систему.

УПРАЖНЕНИЕ. Что бы вы сказали о вычислительной системе, в которой драйвер диска не находится постоянно в памяти?

Открытие файла. Давайте напомним программу, которая открывает на диске файл с именем **IOTEST.DAT**. Мы должны

включить в нее блок из четырех слов, задающий имя файла:

```
FILNAM:      .RAD50      /DK/  
             .RAD50      /IOTESTDAT/
```

Конечно, и одна директива

```
FILNAM:      .RAD50      /DK#IOTESTDAT/
```

(где # обозначает пробел при вводе с клавиатуры) равносильна предыдущему, но она менее ясна. Мониторная программа возьмет имя устройства из первого слова этого блока, имя файла из второго и третьего, а расширение имени файла из последнего. Так, если бы имя файла было **IO.DAT**, мы должны были бы в директиве **.RAD50** написать **/IO####DAT/** с четырьмя пробелами, чтобы до конца заполнить ячейки. Заметьте, что точка, предшествующая расширению имени файла, будет подставлена операционной системой, и ее не нужно включать в директиву **.RAD50**.

На первое слово в блоке **FILNAM** можно, как и ранее, сослаться в мониторном вызове **.FETCH**. Наш блок, однако, строился в расчете на вызов **.ENTER**. Этот вызов предназначен для резервирования места под файл с заданным именем на заданном запоминающем устройстве.

Если файл создан с помощью вызова **.ENTER**, то для ссылки на него указывается не блок **FILNAM**, а просто соответствующий файлу номер. Этот номер присваивается вызовом **.ENTER** и обозначает тот канал, на котором файл был открыт. Используемые в системе RT-11 номера каналов лежат в диапазоне от 0 до 377 (восьмеричное). Канал не является реальным физическим кабелем между машиной и запоминающим устройством: он только идентифицирует номер для конкретных операций ввода-вывода.

Мониторный вызов **.ENTER** имеет пять параметров, разделенных запятыми. Рассмотрим их, несколько нарушив порядок. Пятый параметр необходим для некоторых операций с лентой — его мы оставим пустым. Второй параметр есть выбранный номер канала. Мы можем с равным основанием выбрать, к примеру, первый канал с номером 0 и написать

```
.ENTER      ?,#0,?,?
```

где вместо ? нам предстоит еще что-то подставить. Обратите внимание, что значение 0 в качестве номера канала должно передаваться как обычный аргумент в языке ассемблера. Можно, например, написать

```
CLR          R1  
.ENTER      ?,R1,?,?
```


Нельзя, однако, аналогичным образом употребить регистр **R0**, так как в самом начале **.ENTER** заносит в **R0** указатель на блок из четырех слов; затем этот указатель используется в **EMT**-программе, к которой обращается макро. Эти четыре слова (вдобавок к тем, которые имеют метку **FILNAM**) должны быть завезены программой. Их же можно использовать при вызове любых других программ ввода-вывода. Но некоторые из них требуют более четырех слов. Чтобы иметь запас в таких случаях, а также в расчете на дальнейшие версии системы, обычно пишут

```
IOBLK:      .BLKW      10
```

резервируя восемь слов.

Первый параметр в вызове **.ENTER** должен быть адресом первого слова этого блока. Макро **.ENTER** включит этот блок в список параметров и выполнит команду **EMT**.

Третий параметр в вызове **.ENTER** должен быть адресом блока, в котором специфицируется файл. В результате имеем

```

                                .MCALL      .ENTER
                                ...
START:                        ...
                                ...
                                .ENTER      #IOBLK,#0,#FILNAM,?
                                ...
FILNAM:                        .RAD50      /DK/
                                .RAD50      /IOTESTDAT/
                                ...
IOBLK:                         .BLKW      10
```

Нам осталось рассмотреть последний, четвертый параметр. Он должен определять объем пространства, которое нужно отвести на запоминающем устройстве под создаваемый файл. Если на место этого параметра поставить **—1**, то будет отведено максимально возможное свободное пространство на запоминающем устройстве. В итоге мониторный вызов, который открывает файл **IOTEST.DAT** на диске, будет таким:

```
.ENTER      #IOBLK,#0,#FILNAM,#—1
```

Следующей должна быть команда **BCS** для перехода на фрагмент, который печатает сообщение типа **ENTER FAILED** (файл не открыт) и выполняет другие действия, предусмотренные программистом.

Хотя теперь под файл **IOTEST.DAT** отведено место на диске, монитор пока еще не считает файл постоянным. В частности, элемент, относящийся к этому файлу, не был внесен в каталог

пользователя. В каталоге появится постоянный элемент выполнения мониторного вызова **.CLOSE**, который закрывает файл с указанным номером канала:

.CLOSE #0

Как исключение, этот вызов не устанавливает бит С в случае ошибки, поскольку ее обработкой занимается монитор. Поэтому макровывозов **.CLOSE** не нужно сопровождать командой **BCS**. В макро **.CLOSE**, как во всех других макро ввода-вывода, используется **R0**, содержимое которого поэтому может измениться.

УПРАЖНЕНИЯ. 1. Напишите программу, добавляющую в каталог ваших файлов на диске файл **IOTEST.DAT**. Сколько места он занимает на диске? Что в нем хранится?

2. Воспользуйтесь редактором для записи чего-либо в файл **IOTEST.DAT**. Теперь снова запустите свою программу и посмотрите, что произойдет.

3. Изучите имеющиеся в вашей системе средства для защиты файлов. Защитите файл **IOTEST.DAT**, установив ему статус «только для чтения». После этого снова запустите вашу программу.

Запись в файл. Не будем ограничиваться одним только включением в каталог элемента для пустого файла и вставим между вызовами **.ENTER** и **.CLOSE** команду, которая запишет что-нибудь в файл. Соблюдая сложившуюся традицию, напомним программу, которая создает файл, содержащий букву **B** в коде ASCII.

Чтобы записать в файл, созданный директивой **.ENTER**, воспользуемся макровывозом **.WRITW**. Он имеет пять параметров; первые два из них совершенно такие же, как в вызове **.ENTER**: адрес блока для **EMT**-программы и номер канала, который для этого файла уже указывался в макровывозе **.ENTER**:

.WRITW #IOBLK,#0,?,?,?

В третьем параметре передается адрес, по которому в памяти расположены данные, предназначенные для записи на запоминающее устройство. Если мы где-нибудь в программе заведем слово

MEM: .WORD 102

то сможем записать

.WRITW #IOBLK,#0,#MEM,?,?

Четвертый параметр есть количество слов, которые нужно записать. В нашем случае четвертый параметр в макровывозе **.WRITW** равен 1.

Пятый параметр указывает, в какой блок файла должна производиться запись. В пространстве на диске или магнитной ленте доступны не отдельные слова, а блоки слов. Размер блока является внутрисистемной характеристикой запоминающего устройства. Для диска он равен 256 (десятичным) словам. Мы хотим записать в первый (и только первый) блок нашего файла — блок номер 0. Поэтому макровывозов выглядит так:

.WRITW #IOBLK,#0,#MEM,#1,#0

УПРАЖНЕНИЯ. 1. Допишите программу до конца.

2. Измените ее так, чтобы вместо буквы **B** в файл заносилась буква **C**. Снова пропустите программу. Что произошло с первоначальным файлом **IOTEST.DAT**?

Учтите, что в большинстве систем мониторную команду **TYPE** нельзя использовать для распечатки содержимого файла **IOTEST.DAT**, поскольку она игнорирует текст, не заканчивающийся символом **↵**. Содержимое файла необходимо просматривать с помощью редактора.

Допустим теперь, что нам нужно записать в файл **IOTEST.DAT** весь алфавит. Мы можем сделать это, заносая каждый раз по одному слову. Считая, что вначале ячейка **MEM** содержит число 101, а регистр **R2** очищен, получаем такую программу:

```

                                MOV          #32,R1
LOOP:                          .WRITW      #IOBLK,#0,#MEM,#1,R2
                                BCS          WERR
                                INC          R2
                                INC          MEM
                                SOB          R1,LOOP

```

Номер блока для вызова **.WRITW** содержится в регистре **R2**. Если бы мы не увеличивали содержимое **R2** внутри цикла, то при каждом вызове **.WRITW** запись производилась бы в первый блок нашего файла, причем всякий раз в начало блока. В результате был бы создан файл из одного блока, содержащий только букву **Z** в коде ASCII.

Приведенная программа крайне неэффективно использует место на диске, поскольку создаваемый ею файл содержит двадцать шесть блоков (проверьте по вашему каталогу), в каждый из которых занесено по одной букве. То есть мы заняли более шести тысяч слов дискового пространства для хранения двадцати шести байтов информации. Вместо этого нам следовало с помощью

директивы ASCII поместить весь алфавит в один массив с меткой MEM и уж тогда записать на диск

```
.WRITW      #IOBLK,#0,#MEM,#13.,#0
```

Заметьте, что ассемблер воспринимает число как десятичное, если оно сопровождается точкой; поэтому $13.=^{\wedge}D13=15$.

Директивы повторения. Существует изящный способ, позволяющий сформировать блок с меткой MEM. Он состоит в использовании *макродирективы ассемблера* .IRPC¹⁾. Форма макровывода такова:

```
.IRPC      X,ABCDEFGHIJKLMNOPQRSTUVWXYZ
.ASCII    /X/
ENDM
```

Здесь X — фиктивный параметр, который последовательно замещается литерами, стоящими после запятой в директиве .IRPC. Макрорасширением этой директивы будет такая последовательность:

```
.ASCII    /A/
.ASCII    /B/
```

и т. д. до

```
.ASCII    /Z/
```

Печать расширения такой макро привела бы к чересчур длинному листингу. Если для распечатки макрорасширений применяется директива .LIST ME, то ее действие можно отменить директивой .NLIST ME. Может оказаться удобным перед .IRPC отменить этой директивой печать расширений, а после .IRPC восстановить прежний режим листинга новой директивой .LIST ME.

Укажем также на применение директивы .IRPC в целях генерации последовательности команд, позволяющих переслать содержимое регистров от R0 до R5 в системный стек:

```
.IRPC      X,012345
MOV        R'X, -(SP)
.ENDM
```

Символ ' служит в MACRO-11 для разграничения параметра; без него при трансляции появился бы неопределенный идентификатор RX. При расширении макроассемблер уберет символ

¹⁾ Indefinitely RePeat with Character substitution — повторять неопределенное число раз с заменой литеры, — Прим. перев.

и в нашем случае сгенерирует последовательность команд

```
MOV      R0, -(SP)
MOV      R1, -(SP)
```

и т. д. до

```
MOV      R5, -(SP)
```

Здесь мы могли бы также применить директиву **.IRP**. Вызов выглядел бы так:

```
.IRP      X,<R0,R1,R2,R3,R4,R5>
MOV      X, -(SP)
.ENDM
```

и была бы сгенерирована в точности такая же последовательность команд. Хотя в приведенном примере директива **.IRPC** более корректна, **.IRP** имеет более широкое применение, если требуется подстановка параметра.

Буферизация ввода-вывода. Чтобы бессмысленно не расходовать свободное пространство, программа должна производить запись в запоминающее устройство упаковками, размер которых равен размеру блока данного устройства. Так, если одним вызовом **.WRITW** на диск записывается 256 (десятичных) слов, то тем самым на нем заполняется в точности один блок. Заметьте, что **.WRITW** позволяет записать данные не в произвольное место, а лишь с начала блока диска.

Поэтому нам нужен *буфер* на 400 (восьмеричных) слов. Когда буфер заполнен, он выталкивается на диск. Ячейку **OBUF** отведем для хранения адреса первого слова в буфере вывода. Допустим, что нам нужно записать на диск результаты расчета, которые, к примеру, подпрограмма **CALC** передает по одному числу через регистр **R0**. Следующая последовательность команд заполнит буфер данными и запишет его на диск:

```
MOV      OBUF,R1          ; R1 указывает на начало буфера
MOV      #400,R2          ; R2- счетчик слов
LOOP:    JSR      PC,CALC   ; вычисления
MOV      R0,(R1)+         ; результат в R0
SOB      R2,LOOP
.WRITW   #IOBLK,#0,OBUF,#400,#0
```

Хотя говорят, что буфер *вытолкнут* на диск, данные в нем, конечно, остаются.

Обратите внимание на то, что адрес буфера передается как содержимое ячейки **OBUF**. Если требуется записать более чем один блок, то нам точно так же придется передавать номер блока.

УПРАЖНЕНИЯ. 1. Напишите программу, которая создает на диске файл, содержащий в последовательно расположенных

словах восьмеричные числа:

- а) от 1 до 2000;
- б) от 0 до 2000;
- в) от m до n , где m и n вводятся во время выполнения программы.

2. Напишите программу, которая вводит текст, набираемый на клавиатуре терминала и заканчивающийся символом \$ (выход), и записывает его в файл на диске.

3*. Измените программу из упр. 2 так, чтобы с терминала можно было ввести имя создаваемого файла.

4. Напишите программу, которая создает два файла на диске и вводит текст, набираемый на терминале. Одновременно с вводом текст записывается в первый файл до тех пор, пока не встретится символ ^B; далее текст записывается во второй файл до тех пор, пока не встретится символ ^A, и т. д. (Каждый файл должен иметь свой собственный канал.)

5. Может ли файл быть открыт одновременно на два канала?

На самом деле один макровывод **.WRITW** может заполнить более чем один блок данных. На диск будет записано столько последовательных блоков, сколько необходимо. В программе, которая сразу записывает на диск, скажем, по два блока, нужно между записями увеличивать на 2 номер блока в вызове **.WRITW**.

Чтение файла. Доступ к уже существующему файлу осуществляет макровывод **.LOOKUP**. При этом драйвер соответствующего устройства должен находиться в памяти. Список параметров в **.LOOKUP** аналогичен списку параметров в **.ENTER** с той лишь разницей, что в **.LOOKUP** нет параметра, задающего длину файла. Итак, мы можем получить доступ к файлу, расположенному на нулевом канале, при помощи вызова

.LOOKUP #IOBLK,#0,#FILNAM

где блок **FILNAM** специфицирует файл, а блок **IOBLK**, как и ранее, используется в системных макро.

Получив доступ к файлу, мы можем с помощью мониторного вызова **.READW** считать нужные нам блоки в память. По форме вызов **.READW** аналогичен **.WRITW** с той лишь разницей, что данные пересылаются в противоположном направлении — с диска в буфер. Такая процедура называется *заполнением буфера с диска*.

Можно также обновить файл, применяя макровыводы **.WRITW** после **.LOOKUP**. В некоторых системах это будет работать правильно, если только в файл первоначально были записаны полные блоки.

Во время чтения файла мы должны уметь обнаруживать его конец. При попытке чтения дальше конца файла происходит установка бита *C*, но, поскольку бит переноса может быть установлен и по другим причинам, его проверка не будет надежна. Однако при возврате из **.LOOKUP** в *R0* будет находиться число, равное количеству блоков в том файле, к которому был открыт доступ. Программа должна использовать эту информацию для того, чтобы исключить попытки чтения за пределами данного файла.

УПРАЖНЕНИЯ. 1. Напишите программу, которая в желаемом формате печатает файл, созданный вами в упр. 1 (см. выше) и содержащий восьмеричные числа от 1 до 2000.

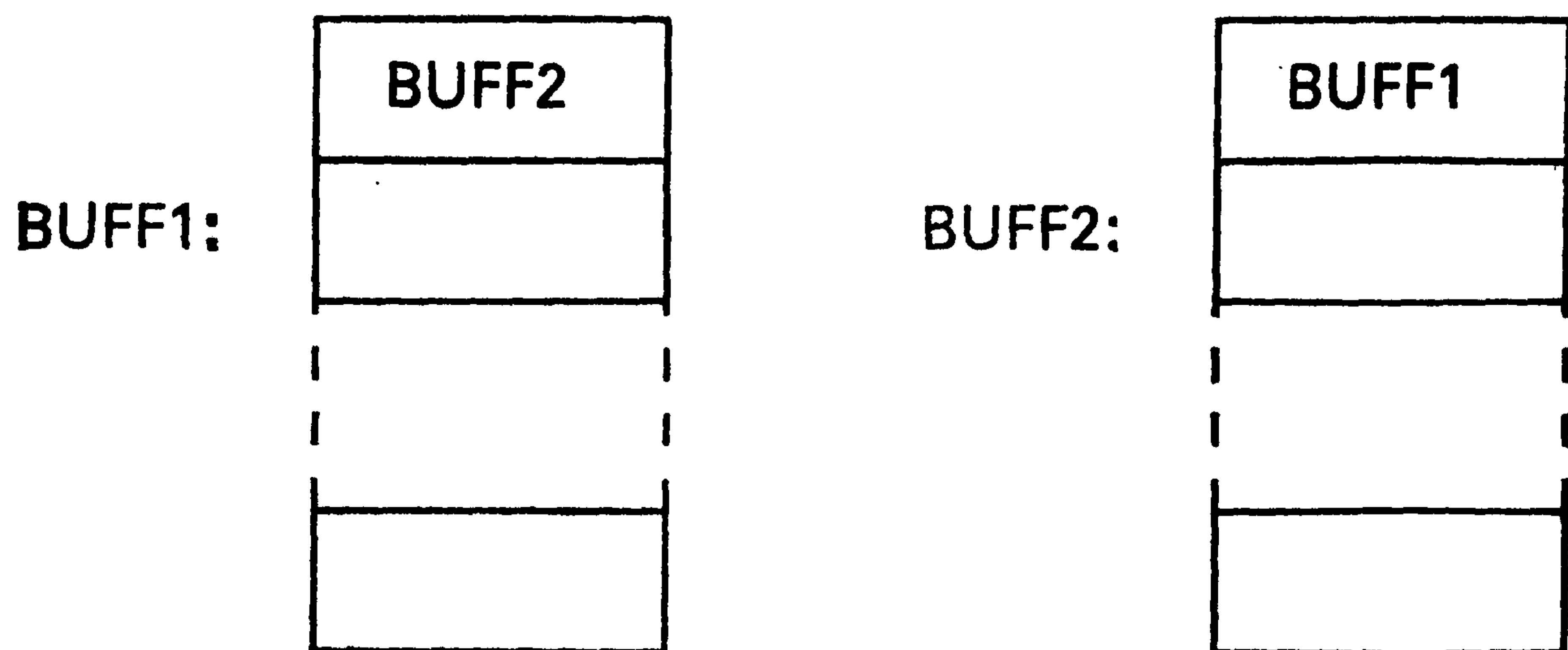
2. Напишите программу, которая заменяет в этом файле числа с 1001 по 1400 на числа с 3001 по 3400, а прочие оставляет прежними.

Многократная буферизация. Макровыводы **.READW** и **.WRITW** возвращают управление программе пользователя только после того, как все необходимые действия будут полностью завершены. Как было замечено в § 4.2, это приводит к непроизводительному расходу машинного времени на ожидание окончания операции ввода-вывода.

Вместо них можно использовать мониторные вызовы **.READ** и **.WRITE** (с тем же синтаксисом). Эти вызовы производят не только подготовку ячеек, управляющих вводом-выводом, но также установку бита, разрешающего прерывания от устройства, и затем немедленный возврат управления программе пользователя. Дальнейшие действия нам знакомы: программа может продолжить свою работу, а устройство, когда оно будет готово, прервет ее.

Рассмотрим еще раз программу, которая заполняет буфер результатами вычислений и затем выталкивает буфер на диск. Не ожидая завершения записи на диск, программа может продолжить вычисления. Однако она не должна затрагивать буфер, который в текущий момент записывается на диск, до тех пор, пока операция ввода-вывода не будет закончена. Поэтому программе нужен еще один буфер, куда заносятся данные в то время, пока первый буфер освобождается. Когда заполнится второй буфер, можно с помощью макровывода **.WRITE** записать его на диск, а программа между тем вновь переключится на первый буфер.

Можно обеспечить плавное переключение буферов, если перед каждым буфером завести *слово-заголовок*, которое содержит ссылку на первое слово другого буфера:



В регистре **R1** будем хранить указатель на первое слово текущего буфера и поэтому начнем с засылки **MOV #BUFF1, R1**. На этот раз первый регистр не должен изменяться внутри цикла вычислений и заполнения буфера. Поэтому, используя регистр **R3** как указатель блока, имеем

```

LOOP:      выполнение вычислений
           и заполнение буфера
           .WRITE      #IOBLK, #0, R1, #400, R3 ; выдача буфера
           INC.        R3                      ; следующий блок
           MOV         -2(R1), R1              ; смена буферов
           BR          LOOP                   ; на продолжение вычислений

```

Заметьте, что фрагмент, в котором выполняются вычисления и результаты заносятся в буфер, даже «не знает», с каким буфером он в данный момент работает, как того и требует истинный дух структурного программирования.

Мы можем применить тот же метод для работы с любым количеством буферов. Заголовок каждого буфера будет указывать на начало следующего, а заголовок последнего — на начало первого. Такая конфигурация называется *кольцом* буферов. Однако не так уж часто получается выигрыш от наличия более чем двух буферов.

В программе необходимо предусмотреть, чтобы вычислительный процесс не опережал операции ввода-вывода настолько, что попытка заполнять буфер возникает прежде, чем завершится запись предыдущего его содержимого в запоминающее устройство. Вычисления не должны обгонять ввод-вывод. (Существует ли обратная проблема?) К сожалению, макровыводы **.READ** и **.WRITE** не сигнализируют о том, что пересылка данных закончена. Для синхронизации этих двух процессов можно воспользоваться мониторным вызовом **.WAIT**. Он имеет один параметр — номер канала — и приостанавливает выполнение программы до тех пор, пока все незавершенные операции ввода-вывода на этом канале не будут закончены.

В тех же целях можно использовать мониторные вызовы

.READC и **.WRITC**. Каждый из них имеет шесть параметров: первые четыре — те же, что и раньше, пятым является адрес *завершающего фрагмента* в программе пользователя и шестым — номер блока. Как и в случае команд **.READ** и **.WRITE**, монитор немедленно возвратит управление программе пользователя. Однако, как только операция ввода-вывода будет завершена, он ответит на прерывание устройства засылкой текущего содержимого счетчика команд в стек и передаст управление завершающему фрагменту, адрес которого указан в макровывозе **.READC** или **.WRITC**.

УПРАЖНЕНИЯ. 1. Напишите программу создания на диске файла, содержащего в подряд идущих словах первые две тысячи простых чисел. Используйте двойную буферизацию.

2. Напишите программу, которая заменяет число в каждом слове заданного файла его наименьшим простым делителем. (Указание: используйте файл, созданный в упр. 1.)

3. Каким образом завершающий фрагмент должен возвращать управление главной программе?

4.4. Внешние запоминающие устройства под управлением пользователя

В настоящем параграфе дается лишь краткое описание процесса управления внешними запоминающими устройствами. Более подробные сведения по этим весьма сложным вопросам выходят за рамки данной книги.

Объектом нашего обсуждения будет дисковая система RK06. Это — широко распространенное современное запоминающее устройство с весьма типичными характеристиками управления. Если у вас имеется соответствующее руководство, то вы сможете проинтерпретировать приводимое ниже описание применительно к любому другому запоминающему устройству, которое есть в вашей системе.

Дисковая система RK06 состоит из шкафа, снабженного устройством управления (контроллером дисководов RK611) и содержащего *пакет дисков*. В системе обычно имеется много пакетов, и пользователь может поставить на дисковод тот из них, который требуется для выполнения текущей задачи. Если снять защитную крышку, то пакет выглядит как две грампластинки (их называют *пластинами*), прикрепленные к шпинделю на расстоянии около 5 см друг от друга. Поверхности пластин, однако, покрыты не воском, а специальным магнитным составом. Во время операций передачи информации они вращаются с частотой 2400, а не $33\frac{1}{3}$ оборота в минуту.

Контроллер дисководов снабжен траверсой с четырьмя выдвижными стержнями, которые размещаются над поверхностями пластин. На конце каждого стержня находится *головка* — нечто аналогичное тому, что имеется на магнитофоне. Схема расположения пластин и головок приведена в верхней части рис. 4.4.

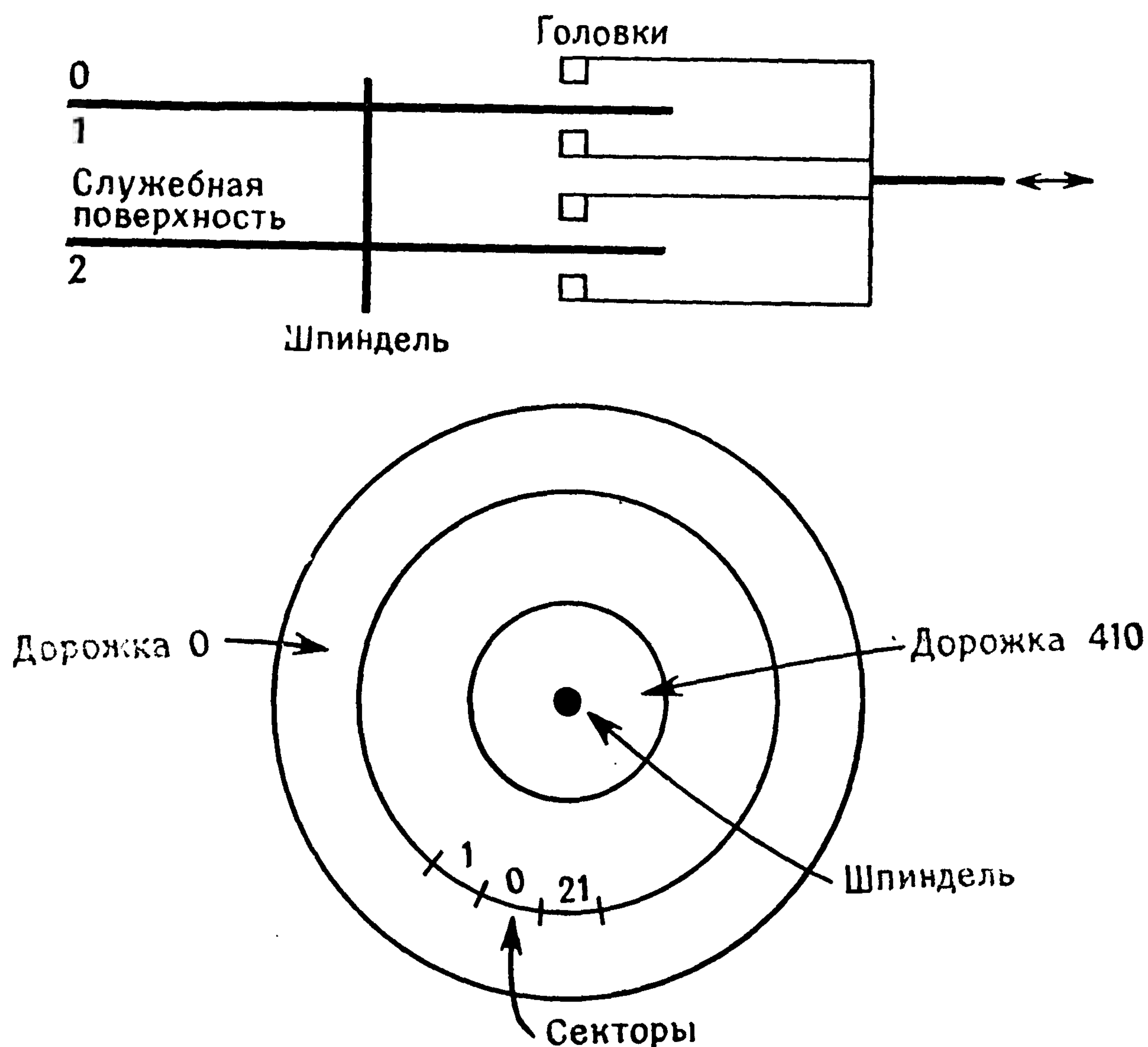


Рис. 4.4. Конфигурация дисковой системы RK06.

Надо иметь в виду, что все четыре головки выдвигаются или втягиваются одновременно.

Одна из четырех поверхностей является служебной (сервоповерхностью): на нее предварительно записывается синхронизирующая и адресная информация, которую потом считывает соответствующая головка. Эта головка предназначена *только для чтения*. Остальные три поверхности доступны для хранения данных пользователя, и соответствующие им головки являются головками *чтения-записи*.

Поверхность пластины разделена на 411 (десятичных) концентрических колец или *дорожек*, причем самая внешняя дорожка имеет номер ноль. Это показано в нижней части рис. 4.4. Несмотря на различие в длинах, каждая дорожка может хранить одинаковый объем информации: **107520** (десятичных) бит.

Обращение к конкретной дорожке протекает в два этапа. Предположим, что нам нужна дорожка с номером 200 на поверхности 1. Во-первых, мы должны приказать контроллеру, чтобы он подвел головки к двухсотой дорожке. Поскольку, однако, все

четыре головки перемещаются совместно, такой приказ задает только «цилиндр» на определенном расстоянии от шпинделя, общий для всех дорожек с номером 200, расположенных на разных поверхностях. Поэтому второй шаг, завершающий спецификацию нужной нам дорожки, заключается в сообщении контроллеру номера поверхности 1.

Регистры устройства. Команды передаются контроллеру через специальные ячейки общей шины, закрепленные за данным запоминающим устройством. Они называются *регистрами устройства*. В большинство из них нельзя ничего занести, манипулируя пультовыми переключателями и переключателем D. Они загружаются командами MOV, BIS или BIC из ЦП. С точки зрения программы ячейки, относящиеся к диску RK06, имеют адреса от 177440 до 177476. Необходимость шестнадцати регистров говорит о сложностях в программировании такого устройства.

Давайте рассмотрим, как подвести головки к двухсотому цилиндру. Если в системе более одного дисковод, то сначала нужно решить, которым из них мы будем управлять. Система позволяет использовать до восьми дисководов, каждому из которых присваивается свой номер (от 0 до 7), указанный на кнопке «готовности». Предположим, что нам требуется накопитель номер 1. Рекомендуем, пока вы не приобретете опыт управления устройством, нажать кнопку «защита записи» на *каждом* дисковом, если стоящий на нем пакет содержит хоть какую-нибудь полезную информацию.

Номер дисковода определяется комбинацией битов 0 — 2 в *регистре управления и состояния номер 2* системы RK06; адрес регистра — 177450, а мнемоническое обозначение — RKCS2. Однако, перед тем как что-либо записывать в регистр диска, необходимо проверить готовность контроллера к приему команд. Контроллер сообщает о своей готовности, устанавливая седьмой бит *регистра управления и состояния номер 1*: RKCS1=177440. Таким образом, выбор накопителя 1 осуществляется так:

	MOV	#RKCS1,R1
1\$:	TSTB	(R1)
	BPL	1\$
	MOV	#1,RKCS2

Обратите внимание на то, что *содержимое регистров устройства может быть изменено только командами, оперирующими словами*, а не байтами.

Следующий шаг состоит в посылке контроллеру сообщения о «подтверждении запроса», которое говорит о «желании» процессора передавать команды. Сообщения и команды посылаются контроллеру установкой битов регистра RKCS1, представляю-

щих код соответствующей функции. Подтверждение запроса имеет код 3. Контроллер ответит установкой бита готовности (который сбрасывается при посылке сообщения о подтверждении запроса). До тех пор, пока этот ответ не будет получен, программа не может продолжать операции управления диском.

```

2$:      MOV      #3,(R1) ; R1 все еще указывает на RKCS1
          TSTB    (R1)
          BPL     2$

```

Убедившись в готовности контроллера, программа должна еще проверить готовность дисководов. Одно не следует из другого: контроллер может быть готов, в то время как, к примеру, головки продолжают двигаться в ответ на предшествующую команду. Для проверки мы должны обратиться к регистру *состояния дисковода* **RKDS=177452**. Перед тем как можно будет продолжить выполнение программы, необходимо, чтобы четыре бита этого регистра были установлены (самим устройством, так как этот регистр только для чтения). Бит 15 устанавливается для указания на то, что регистр содержит текущую информацию для выбранного дисковода. Поскольку в последний раз информация занесена в регистр **RKDS** в ответ на сообщение о подтверждении запроса, этот бит действительно должен быть установлен. Биты 0 и 7 отмечают соответственно, что дисковод доступен контроллеру и готов принимать команды. Шестой бит устанавливается в ответ на подтверждение запроса и служит признаком того, что техническая сторона дела в порядке. Пока все эти биты не окажутся установленными, последующие команды выполняться не будут. А до тех пор программа должна ждать.

```

MOV      RKDS,R2
COM      R2
BIT      #100301,R2
BNE      1$           ; повторить сначала

```

Перед командой **BIT** необходимо инвертировать биты: непосредственная проверка **BIT #100301, RKDS**, за которой следует **BEQ 1\$**, приведет к тому, что все окажется нормальным, даже если установлен только один из четырех битов. Некоторые ошибочные ситуации могут приводить к сбросу битов выборки дисковода в **RKCS2**. Поэтому по техническим причинам перед очередной проверкой готовности дисковода может потребоваться повторное подтверждение запроса. Из-за этого мы передаем управление на начало нашей программы работы с устройством — метку **1\$**.

Вся процедура, таким образом, должна повторяться перед каждой новой операцией с диском. Будем считать, что она оформ-

лена в виде подпрограммы **CHKDSK** и использует **PC** как регистр связи.

После возврата из подпрограммы **CHKDSK** можно послать команду контроллеру диска, зная, что теперь ничто не препятствует ее исполнению (дискуссию по поводу возможных ошибок отнесем в конец параграфа). Чтобы подвести головки к двухсотому цилиндру, надо сначала установить *регистр требуемого цилиндра* **RKDC=177460**:

MOV #200.,RKDC

Движение головок происходит после записи в регистр **RKCS1** кода 17, соответствующего команде «поиска»:

MOV #17,RKCS1

Более точно, команда поиска кодируется числом 16. Нулевой бит **RKCS1** является *пусковым битом*: для того чтобы закодированная в битах 1—4 команда начала выполняться, он должен быть установлен. Как только контроллер диска воспримет эту команду, аппаратура сделает все необходимое, и от программы более ничего не потребуется.

Разметка. Каждая дорожка на пластине диска разделена на доступные по отдельности блоки хранимых данных, называемые *секторами*. Это показано в нижней части диаграммы на рис. 4.4. В отличие от разбиения пластины на 411 дорожек деление дорожки на секторы не является физической характеристикой устройства. Напротив, это есть результат *разметки* поверхности, осуществляемой специальной программой. Некоторые диски (например, гибкие диски) продаются уже размеченными, но диск **RK06** должен быть размечен пользователем. Процесс разметки слишком сложен, чтобы быть предметом обсуждения в этой книге (программа операционной системы DEC, которая выполняет эту работу, содержит более шести тысяч строк), поэтому мы вынуждены допустить, как оно и есть в большинстве случаев, что входящие в состав вашей системы диски уже размечены. Если диск оказался неразмеченным, то операции чтения или записи на нем не выполнимы.

На размеченном диске каждая дорожка разбита на 22 сектора. Каждый сектор содержит три слова *заголовка* и 256 шестнадцатиразрядных слов данных. Другой способ разметки допускает восемнадцатиразрядные слова данных, но при этом на дорожке размещается лишь 20 секторов. Между заголовками и данными, а также между секторами существуют пустые промежутки,

позволяющие управляющему устройству отличать их одно от другого.

В заголовок каждого сектора записываются номера цилиндра, поверхности и сектора. В нем не содержится какой-либо информации, касающейся имени файла, и, чтобы установить соответствие между именами файлов и адресами на диске, требуется системная программа. Если такой программы нет, то приходится самому помнить о том, где что хранится.

Запись на диск и чтение с диска. Допустим, к примеру, что нам нужно записать в тринадцатый сектор сотой дорожки поверхности 1. Заполним 256 слов этого сектора числами от 1 до 400 (восьмеричное). В программе нужно завести буфер, начинающийся, положим, с метки **OBUF**, и заполнить ячейки с **OBUF** по **OBUF+776** числами от 1 до 400.

Пересылка информации между памятью и диском возможна, лишь когда диск вращается. Некоторые диски устроены так, что вращаются непрерывно; однако в системе RK06 по ряду причин допускается остановка шпинделя. Поэтому вначале нужно дать команду запустить шпиндель, которой соответствует код 10 в **RKCS1**:

JSR	PC,CHKDSK	; ожидание готовности диска
MOV	#11,RKCS1	; включение шпинделя

Пока диск не раскрутится, следующее обращение к подпрограмме **CHKDSK** может привести к многократному повторению цикла ожидания готовности, поскольку, пока шпиндель не наберет полную скорость, контроллер не установит бит готовности дисководов.

Далее для выполнения команды записи в регистры устройства должна быть занесена соответствующая информация. В то же время до установки пускового бита их изменять нельзя. В результате мы имеем выбор между такими действиями:

1. Опрос пускового бита, пока он не будет сброшен контроллером.

2. Опрос бита готовности контроллера, пока он им не будет установлен. Контроллер очищает этот бит, когда программа устанавливает пусковой бит, и наоборот.

3. Вызов подпрограммы **CHKDSK**, в которой предусмотрен опрос из п. 2.

Для простоты мы выберем третий вариант. На выходе и контроллер, и дисковод должны быть готовы, а потому разрешено как устанавливать регистры, так и обращаться к команде записи. Регистр **RKDC** должен указывать на сотый цилиндр:

MOV	#100.,RKDC
-----	------------

Поверхность и номер сектора задаются в *регистре адреса диска* **RKDA=177446**. Номер поверхности (в нашем случае 1) записывается в старший байт, а номер сектора (у нас 13) — в младший. Не забывайте, однако, что для занесения в **RKDA** байтовые команды не годятся.

В *регистр адреса шины* **RKBA=177444** нужно занести адрес, с которого начинаются пересылаемые данные. В связи с этим вам нужно познакомиться с приведенными ниже замечаниями по поводу перемещаемых адресов.

В *регистр счетчика слов* **RKWC=177442** нужно занести количество пересылаемых слов, причем в него записывается *двоичное дополнение* к числу слов. В нашем случае нужно переслать 400 слов, поэтому в **RKWC** заносим число —400.

Наконец, нужно дать команду «запись», которой соответствует код 22 в **RKCS1**, и не забыть при этом установить пусковой бит. Вся программа, начиная с включения шпинделя, такова:

JSR	PC,CHKDSK	; ожидание раскручивания диска
MOV	#100.,RKDC	; цилиндр 100
MOV	#1,R0	; поверхность 1
SWAB	R0	; в старший байт
ADD	#13.,R0	; сектор 13
MOV	R0,RKDA	
MOV	#OBUF,RKBA	; буфер вывода
MOV	#-400,RKWC	; счетчик слов
MOV	#23,RKCS1	; запись

Заметьте, что мы не поставили команду поиска перед записью на диск. Команда записи уже включает поиск требуемого цилиндра. Когда поиск завершен, расположенная над заданной поверхностью головка начинает искать нужный сектор на выбранной дорожке (проекция сотого цилиндра на поверхность 1 и есть сотая дорожка на этой поверхности). Когда в результате проверки заголовков сектор будет найден, начнется пересылка информации. Начиная с указанного в **RKBA** адреса, слова поочередно записываются из оперативной памяти в сектор, причем для каждой команды записи пересылка осуществляется, начиная с первого слова сектора. Вся эта последовательность действий выполняется контроллером в ответ на команду записи без какого-либо дополнительного вмешательства программы.

После записи очередного слова контроллер увеличивает на 1 содержимое регистра **RKWC**, чтобы вести счет оставшимся словам, и увеличивает на 2 содержимое регистра **RKBA**, чтобы он указывал на следующую ячейку памяти, откуда предстоит переслать данные. Как только в регистре **RKWC** появляется нуль, контроллер прекращает пересылку данных из памяти. Если в этот момент текущий сектор до конца еще не заполнен,

в оставшуюся часть записываются нули, причем содержимое регистров **RKWC** и **RKBA** остается прежним.

Каждый раз как заполнен очередной сектор, контроллер увеличивает младший байт регистра **RKDA**, чтобы он указывал на следующий сектор. Если при этом младший байт **RKDA** уже равен 21 (десятичное), то контроллер сбрасывает его и увеличивает на 1 старший байт, в результате чего **RKDA** будет указывать на первый сектор той же дорожки на следующей поверхности.

Если **RKDA** уже указывает на двадцать первый сектор второй поверхности, контроллер очистит **RKDA** и увеличит **RKDC**. Смысл именно такого порядка заполнения секторов состоит в том, чтобы как можно реже выполнять отнимающую много времени операцию перемещения головок.

После перехода к новому сектору, как это только что было описано, контроллер проверяет счетчик слов **RKWC**. Если данные еще остались, он записывает их в этот сектор, и весь процесс повторяется.

Если в конце сектора значения **RKWC** наконец окажется равным нулю, контроллер установит бит готовности контроллера и закончит работу. Заметьте, что, согласно описанной выше схеме, регистры **RKDA** и **RKDC** при этом указывают на сектор, следующий за последним записанным.

Чтение с диска абсолютно аналогично записи на диск. Команде чтения соответствует код 20 в **RKCS1**.

Перемещаемые адреса. Этот пункт станет понятен только после прочтения следующего параграфа, но для полноты изложения он приводится здесь.

Содержимое регистра **RKBA** определяет 16 младших битов адреса общей шины, с которого начинается пересылка данных при работе с диском. Биты 8 и 9 в **RKCS1** устанавливаются программой в соответствии с требуемыми значениями битов 16 и 17 адреса общей шины для пересылаемых данных. Если биты 8 и 9 в **RKCS1** равны 0, то **RKBA** по-прежнему определяет физический (в котором биты 16 и 17 равны нулю), а не виртуальный адрес.

Так, для того чтобы подготовить к прочтению в буфер вывода терминала одного слова с диска (невероятная затея), недостаточно занести в **RKBA** код 177566: нужно также установить биты 8 и 9 **RKCS1**. В общем случае этими битами занимается программа, которая по ссылке на соответствующий регистр адреса страницы реализует программный вариант функции управления памятью.

Прерывания. Разряд разрешения прерываний от диска RK06 есть шестой бит в регистре **RKCS1**. Установка этого бита бес-

смысленна после того, как дана команда, даже если в этот момент устройство управления еще продолжает ее выполнять. Программа должна устанавливать бит разрешения прерывания одновременно с посылкой команды. Таким образом, действие

MOV #123,RKCS1

закljučается в следующем: запись на диск и прерывание после завершения команды.

Приоритет прерываний от диска RK06 равен пяти, а адрес вектора прерываний 210. В ячейку 210 должен быть помещен адрес составленной пользователем программы обработки прерываний, а ячейка 212 должна содержать номер приоритета, под которым ЦП будет выполнять эту программу. По причинам, о которых будет сказано ниже, диск может прерывать даже свою собственную программу обработки прерываний. Поэтому, чтобы не потерять управления, рекомендуем в ячейку 212 заносить по крайней мере пятый приоритет.

Значительно осложняет управление диском то обстоятельство, что существуют три различные причины прерываний: готовность дисководов, готовность контроллера и ошибка.

Прерывания от дисководов и контроллера. Как дисковод, так и контроллер могут вызвать прерывание по завершении своего индивидуального задания. Как только возникло прерывание от контроллера, программа может начать загрузку регистров устройства, подготавливая их к следующей команде, но дать команду нельзя до тех пор, пока нет готовности дисководов. Поэтому иногда важно знать, что прерывание вызвано дисководом, а не контроллером. С этой целью программа может проверить бит 14 в регистре RKCS1, который устанавливается в случае прерывания от дисководов. Бит прерывания от дисководов можно сбросить и тем самым подготовить следующее прерывание, если выдать команду «очистить дисковод накопителя», которой соответствует код 4 в RKCS1. Не забудьте при этом установить пусковой бит.

Бесполезно опрос бита прерывания дисководов производить в программе обработки прерываний. Если после прерывания от контроллера обрабатываемая его программа выполняется с высоким приоритетом, дисковод не сможет сообщить о своем прерывании, и программа заикнется. Однако если известно, что программа обрабатывает прерывание именно от контроллера, то она могла бы среагировать на это, перейдя в цикл опроса бита готовности дисководов (в регистре RKDS).

То же самое можно осуществить, заведя в программе ячейку DSKRDY и засылая в нее, скажем, нуль, если дисковод занят выполнением команды, и —1, если он готов к новой. При таком

подходе подпрограмма **CHKDSK** должна начинаться так:

```
CHKDSK:    TST      DSKRDY
           BPL      CHKDSK
```

Когда диск будет готов, программа выйдет из этого цикла и продолжит серию проверок, о которых шла речь выше. Если все они окажутся удачными, произойдет возврат из подпрограммы **CHKDSK**, и вызывающая программа сможет послать свою команду. Перед этим, однако, она должна очистить ячейку **DSKRDY (CLR DSKRDY)**, отмечая тем самым, что устройство занято.

В момент готовности дисководов произойдет прерывание подпрограммы **CHKDSK**, а программа обработки прерывания установит нужное значение ячейки **DSKRDY**, отмечая, что дисковод свободен, и после этого возвратит управление.

```
SERV:      BIT      #40000,RKCS1    ; от чего прерывание ?
           BEQ      1$              ; от контроллера: жди готовности дисководов
           DEC      DSKRDY          ; прерывание от дисководов
           BIC      #100100,RKCS1   ; запрещение прерываний
1$:         RTI
```

(Почему в этом фрагменте не проверяется, не стало ли значение ячейки **DSKRDY** равным **—1**?) Как уже отмечалось, после обработки прерывания от дисководов программа должна запрещать дальнейшие прерывания. Однако, казалось бы, естественное для достижения этой цели применение команды **BIC #100, RKCS1** имеет нежелательный побочный эффект. По техническим причинам эта команда вызывает также установку бита сброса контроллера (бит 15 в **RKCS1**), что приводит к сбрасыванию всех регистров в системе **RK06** (за исключением битов, сигнализирующих об ошибках дисководов); в результате будут утеряны все следы о текущих адресах обмена. Чтобы этого не произошло, используется специальный запрос контроллеру на очистку бита сброса, что и сделано в приведенной выше программе.

УПРАЖНЕНИЯ. 1. Нарисуйте блок-схему программы управления диском, используя подпрограммы, аналогичные **CHKDSK** и программе обработки прерываний.

2. Напишите программу, чтобы запустить остановленный шпиндель; запустите его, потом остановите (команда «разгрузки» — ей соответствует код **6** в **RKCS1**); затем запустите снова.

3. Напишите свои собственные макро для включения и выключения шпинделя диска. (Если команда выдается, когда шпиндель уже находится в требуемом состоянии, контроллер вызовет прерывание, а дисковод — нет).

Ошибки. Самую неприятную проблему в программном управлении запоминающим устройством мы приберегли напоследок.

К сожалению, объем книги позволяет лишь слегка коснуться этой темы.

Если возникает ошибка, то в зависимости от ее характера контроллер установит соответствующий бит в одном из регистров устройства. Он также установит бит 15 в **RKCS1**, который, являясь битом контроллера, в то же время есть *объединенный бит ошибки*. После этого контроллер вызовет прерывания, если они ему разрешены.

Следовательно, прежде чем выполнять иные функции, программа обработки должна проверить, не вызвано ли прерывание ошибкой:

SERV:	TST	RKCS1	
	BMI	IOERR	: программа обработки ошибок

Ответственность за некоторые ошибки лежит на программисте. Например, контроллер в процессе передачи информации может достичь конца двадцать первого сектора второй поверхности 410-го цилиндра, а данные еще не исчерпаны. Такая ситуация приведет к установке девятого бита в *регистре ошибок дискового да*, **RKER=177454**.

В то же время множество ошибок возникает как естественное следствие, когда делается попытка установить связь между такими сложными и в корне различными устройствами, как диск и память. Особенно распространены ошибки синхронизации. Если проверка показывает, что подобная «аппаратная ошибка» привела к установке бита 15 в **RKCS1**, то лучше всего попытаться повторить требуемую операцию. Нужно сбросить указанный бит, не забывая, что это приведет к обнулению всех регистров устройства, так что, если их содержимое имеет значение, нужно позаботиться об их сохранении. Потом нужно выдать команду **RESET** для инициализации общей шины и повторить операцию ввода-вывода.

УПРАЖНЕНИЯ. 1. При помощи имеющегося в вашем распоряжении руководства составьте полный перечень возможных ошибок вашего запоминающего устройства и соответствующих им битов.

2 *. Напишите программу для пересылки данных из одного места памяти в другое через диск.

4.5. Управление памятью

Обычная интерпретация машинного слова PDP-11 как адреса позволяет нам адресоваться к байтам в диапазоне от 0 до 177777 т. е. всего к $2^{16}=65536$ байтам. При описании различных объемов памяти принято пользоваться обозначением К, которое соответст-

вует числу $2^{10}=1024(=0\ 2000)$. Поэтому можно сказать, что одно слово машины PDP-11 позволяет нам адресоваться к полю памяти объемом 64К байт или 32К слов.

В стандартном оборудовании каждого процессора серии PDP-11 предусмотрен определенный объем памяти, который в случае необходимости можно расширить с помощью имеющихся в аппаратуре возможностей. Было бы естественно считать, что максимальный объем памяти для машины PDP-11 равен 32К слов, просто потому, что для большего не хватает адресного поля. Так как 4К слов общей шины зарезервированы под регистры ввода-вывода и различные управляющие функции, то, казалось бы, система PDP-11 предоставляет в распоряжение не более чем 28К слов оперативной памяти. Для самых малых машин данного семейства это действительно так.

Однако в более крупных системах PDP-11 внутренние регистры ЦП имеют восемнадцать разрядов, так же как и адреса общей шины. Поэтому ЦП может адресовать, а общая шина обеспечивать доступ к пространству в $2^{18}=256К$ байт или 128К слов. Учитывая 4К слов, отводимые под регистры ввода-вывода и аналогичные средства, потенциально получаем 124К слов оперативной памяти. Казалось бы, это огромный объем, но некоторые пользователи (с непомерными запросами), работающие в системах с разделением времени, умудряются его полностью исчерпывать.

Указанная память, однако, составлена из уже знакомых нам шестнадцатиразрядных слов машины, и, казалось бы, проблема, по-прежнему остается неразрешимой, поскольку одно слово из шестнадцати битов не может вместить восемнадцатиразрядный адрес. Если пользователь не предпримет никаких мер, то так оно и будет: только 32К адресов общей шины будут доступны его программе. Из них 28К — ячейки оперативной памяти. Так как $56К=0\ 160000$ (в байтах), то обычным способом можно программно сослаться на ячейки от 0 до 157776. Ссылка интерпретируется процессором обычным образом: как обозначение ячейки общей шины с указанным номером. К примеру, команда **CLR@#100000** очищает ячейку общей шины с номером 100000. Эта ячейка является одним из слов оперативной памяти. В данном параграфе мы будем использовать абсолютную адресацию, поскольку значение адреса представляется при таком способе в явном виде.

Регистры устройств ввода-вывода, однако, всегда находятся в верхних адресах общей шины и занимают 4К слов, т. е. в случае шестнадцатиразрядных адресов это ячейки с 760000 по 777776. Например, буфер печатающего устройства занимает ячейку с адресом 777566. Такое число не помещается в одно слово машины PDP-11, из-за чего команду типа **MOVB @#102,777566**

закодировать без изменения нельзя. Тем не менее мы уже видели, что запись **MOVB #102, @#177566** позволяет адресоваться к требуемому буферу. Происходит так потому, что ЦП автоматически *смещает* значения всех адресов в диапазоне от 160000 до 177776, интерпретируя их как шестнадцать младших битов истинного адреса, в котором 17-й и 18-й биты равны нулю.

Очень важно не путать подобное смещение адресов с тем, которое производит компоновщик. Задача последнего состоит в том, чтобы вне зависимости от места размещения программы в памяти исполнительный адрес, вычисляемый процессором, соответствовал требуемой ячейке памяти. К примеру, если мы пишем **MOVB #102, 177566**, то от компоновщика требуется определенная работа, в результате которой он убедится, что здесь имеется ссылка на ту же ячейку памяти, что и при записи **MOVB #102, @#177566**. В обоих вариантах ссылаются на ячейку 177566. В функцию смещения, о которой идет речь в данном параграфе, входит привязка этого адреса (*виртуального* адреса) к соответствующей ячейке на общей шине (*физическому* адресу). Обратите внимание также на то, что преобразование виртуального адреса в физический выполняется аппаратно.

Ясно, что в диапазоне от 0 до 157776 никакого смещения не требуется: виртуальный адрес совпадает с физическим. Однако для виртуальных адресов с 160000 по 177776 физический адрес получается расширением виртуального адреса двумя единичными битами: разрядами семнадцатым и восемнадцатым.

Машины PDP-11 с памятью более чем 28К слов имеют специальный блок проверки величины смещения. При соответствующем подборе виртуального адреса и значения смещения любой физический адрес оказывается доступным. Аппаратура, при помощи которой осуществляется контроль смещения, называется *блоком управления памятью*. На некоторых машинах PDP-11 он встроен в центральный процессор, на других является дополнительным оборудованием, а на самых малых вообще отсутствует.

В различных моделях ЦП управление памятью различается по уровню сложности. Наше обсуждение будет сосредоточено на модели 11/34, в которой управление памятью есть неотъемлемая часть ЦП. Попутно будут упоминаться особенности других моделей.

Страничная организация памяти. На машине PDP-11/34 пространство виртуальных адресов разбито на восемь *страниц*, каждая из которых рассматривается блоком управления памятью, как единое целое. Имеется 32К слов виртуальных или программных адресов, а длина каждой страницы равна 4К слов.

Привязка виртуальных адресов к страницам определена заранее и не может быть изменена. Конкретно она выглядит так:

Номер страницы	Диапазон виртуальных адресов
0	000000-017776
1	020000-037776
2	040000-057776
3	060000-077776
4	100000-117776
5	120000-137776
6	140000-157776
7	160000-177776

С каждой страницей связаны две специальные ячейки общей шины: *регистр адреса страницы* и *регистр описания страницы*. Будем использовать для них мнемонику: **KPAR0—KPAR7** и **KPDR0—KPDR7**; выбор буквы **K** будет объяснен позднее. Соответствие между регистрами и ячейками (физическими) общей шины такое:

Номер страницы	KPAR	KPDR
0	772340	772300
1	772342	772302
2	772344	772304
3	772346	772306
4	772350	772310
5	772352	772312
6	772354	772314
7	772356	772316

Если бы процессор модели 11/34 был установлен на изолированной системе, указанные ячейки были бы доступны с пульта оператора. К ним также можно обращаться из программы постольку, поскольку в этом есть потребность. Мы уже видели, что, даже если блока управления памятью не существует, седьмая страница всегда размещается так, чтобы эти (физические) адреса оказались доступными.

Если же машина PDP-11/34 работает в режиме разделения времени, а вы не привилегированный пользователь, то вам не удастся добраться до этих ячеек. Позже в этом параграфе мы изучим прием, позволяющий осуществить такую защиту.

Включение управления памятью. При введении в действие процессора или при перезапуске его после команды **HALT** блок управления памятью не работает. Он активизируется установкой бита включения управления памятью, а именно нулевого бита регистра состояния управления памятью номер 0; мнемоническое обозначение — **SR0**, адрес на общей шине 777572.

Заметьте, что, пока управление памятью выключено, автоматическое подключение седьмой страницы позволяет нам обращаться к регистру **SR0** как к виртуальному адресу **177572**. Поэтому включение блока управления памятью достигается так:

SR0=177572

...
BIS **#1,@#SR0**

Рассмотрим теперь такую последовательность команд:

SR0=177572

KPAR7=172356

...
CLR **KPAR7**
BIS **#1,@#SR0**
BIC **#1,@#SR0**

Если машина только что включена, то нет необходимости сбрасывать регистр **KPAR7**, так как он заведомо содержит нуль. Но если процессор остановлен после функционирования операционной системы, то, несмотря на отключение блока управления памятью, регистры продолжают сохранять установленные в них системой значения. При нормальной работе блок управления памятью запомнит содержимое регистра **KPAR7**, так что седьмая страница будет, как и обычно, отведена под регистры устройств ввода-вывода. Для этого требуется специальным образом настроить упомянутый регистр, содержимое которого было вытеснено нашей командой.

Нужно помнить, однако, что сбрасывание битов регистра производилось при отключенном блоке управления памятью. Поэтому ссылка в следующей проверке на регистр **SR0** как на виртуальный адрес **177572** аппаратно приводит к физической ячейке с адресом **777572**. Следовательно, эффект от выполнения команды **BIS** сохраняется.

Следующей командой **BIC** мы попытались вновь отключить блок управления. Именно к такому результату привело бы сбрасывание первого бита ячейки **777572** общей шины. Но в нашей команде мы адресуемся к **177572**. Предыдущая команда привела к отключению автоматической привязки виртуального адреса **177572** к физической ячейке **777572**. Поэтому аппаратура, заметив, что **177572** есть виртуальный адрес в седьмой странице, обратится к регистру **KPAR7**, чтобы определить, как его надо преобразовать. Мы же намеренно воздержались от записи в регистр **KPAR7** необходимого смещения. В итоге команда **BIC** обратится не к регистру состояния, а к некоторой другой физической ячейке памяти и не выполнит возложенную на нее функцию.

Не существует иного способа программно исправить сло-

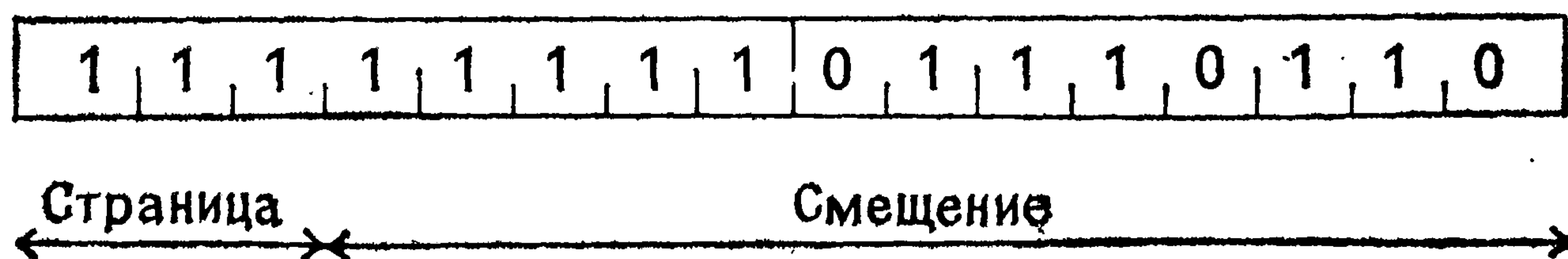
жившуюся ситуацию. Поскольку невозможно обратиться к регистру **KPAR7**, нельзя задать и правильное смещение. По аналогичной причине все регистры устройств ввода-вывода оказываются программно недостижимыми. Единственный способ отключить блок управления памятью состоит в остановке процессора.

Формирование физического адреса. При включенном блоке управления памятью каждая ячейка, к которой адресуются из программы, рассматривается аппаратурой как состоящая из двух полей. Биты с 13-го по 15-й определяют номер страницы виртуального адреса. Это так называемое *поле текущей страницы*. Пусть виртуальный адрес равен 177566. Биты с 13-го по 15-й в нем равны 1, что соответствует двоичной записи числа 7, т. е. данный адрес задает седьмую страницу. Поэтому аппаратура обращается к регистру **KPAR7** за *базовым адресом* страницы 7. Он кодируется битами с 0-го по 11-й регистра адреса страницы (остальные биты в модели 11/34 не используются). Для получения действительного значения смещения это двенадцатирядное поле сдвигается на 6 битов влево.

Оставшиеся 13 битов виртуального адреса (с 0-го по 12-й) представляют собой *поле смещения* внутри данной страницы. Значение этого поля складывается с базовым адресом, и в результате формируется физический адрес, соответствующий данному виртуальному.

Обратите внимание на то, что все действия выполняются аппаратно. Программисту необходимо только настроить регистры.

Обычно операционная система заносит в регистр **KPAR7** число 7600. Пусть опять виртуальный (программный) адрес равен 177566. Он распадается на два поля:

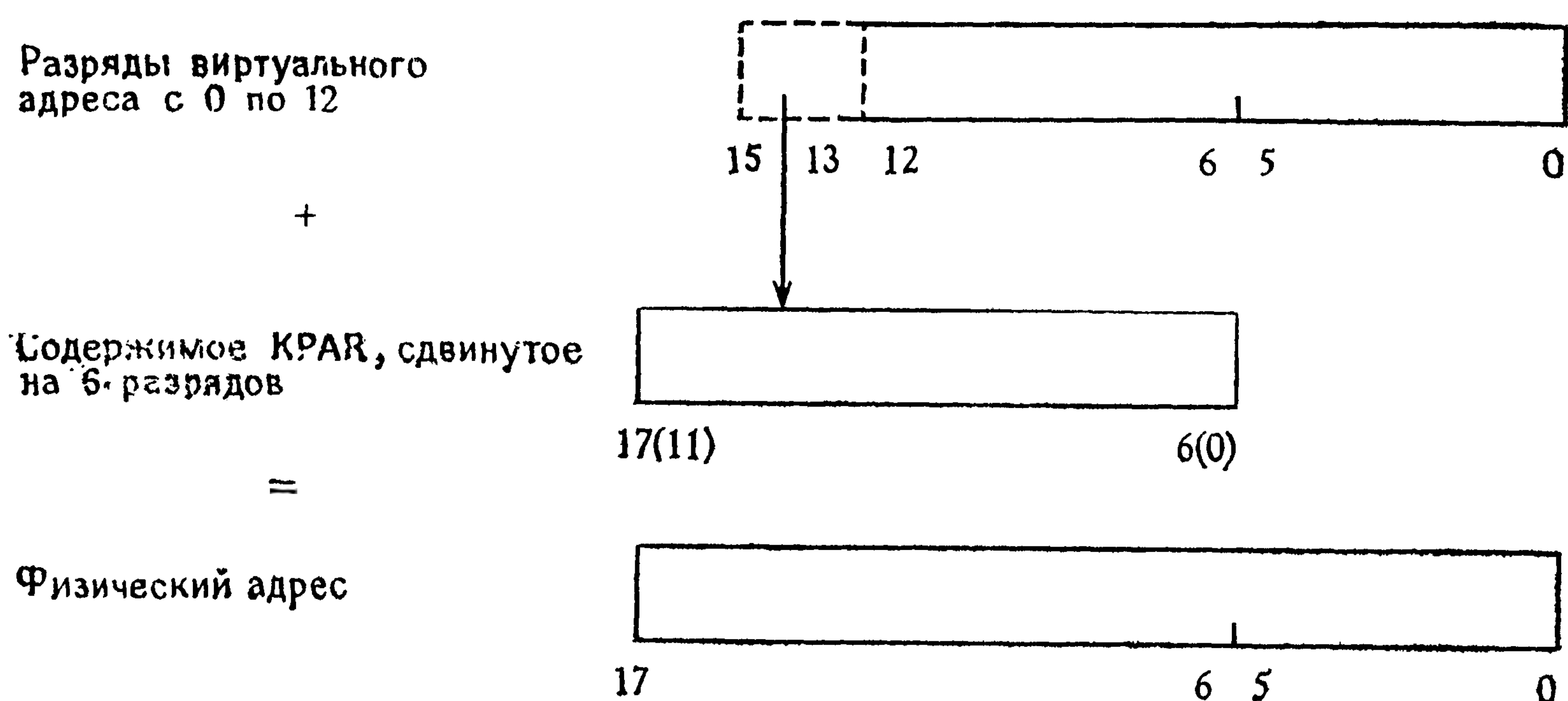


которые определяют седьмую страницу и смещение 17566. Отметим, что смещение всегда задает адрес (в байтах) между 0 и 17777, так как длина страницы составляет 8К (= 0 2000) байтов или 4К слов. Сместя содержимое регистра **KPAR7** (т. е. 7600) на 6 битов влево, получим базовый адрес данной страницы, равный 760000. Следовательно, физическим адресом, соответствующим данному виртуальному, будет

+ Базовый адрес	+ 760000
+ Смещение	+ 17566
Физический адрес	777566

как и должно быть для буфера печати терминала.

На диаграмме описанный процесс формирования адреса можно изобразить так:



Отсюда видно, что две младшие восьмеричные цифры (биты с 0-го по 5-й) всегда одинаковы для физического и виртуального адресов. Прибавление к содержимому **КРАР** единицы приводит к увеличению физического адреса на 100. Порция в 100 байтов есть, следовательно, наименьшая величина смещения, и поэтому в блоке управления памятью она служит *квантом*.

Размер страницы и доступ к ней. Страница виртуальной памяти всегда содержит 200 квантов. Блок управления памятью распределит первый квант страницы, вычислив базовый адрес по описанной выше схеме. Но он не будет выполнять ту же операцию с остальными квантами до особой команды. В результате адресация к ячейкам с 000100 по 017776 не достигнет цели, потому что им не будут поставлены в соответствие реальные ячейки памяти. Ссылка же на адреса с 020000 по 020076 будет реализована.

Чтобы задать количество квантов данной страницы виртуальных адресов, которое должно быть приведено к физическим ячейкам, необходимо обратиться к регистру описания страницы. В его старший байт программист заносит требуемое число квантов (помимо первого). По команде

BIS

#77400,@#KPDR7

блок управления памятью узнает о том, что все 200 квантов седьмой страницы должны преобразоваться в физические адреса. Команда задает *размер* страницы в 200 квантов.

Первый и второй биты регистра данных **KPDR** каждой страницы должны устанавливаться программистом, чтобы определить *права доступа* процессора к ней. Если они оба равны 1, то разрешаются любые обращения из программы к соответствующим

щей виртуальной памяти; это *резидентная*, открытая на *чтение-запись* страница. Если в первом бите 1, а во втором 0, то ссылки разрешены только в командах, которые не изменяют содержимого ячеек памяти; это *резидентная*, открытая *только на чтение* страница. Если, наконец, оба бита нулевые, программная адресация к данной странице запрещена: это *нерезидентная* страница.

Таким образом, последовательность команд

```
MOV      #7600,@#KPAR7
BIC      #6,@#KPDR7
BIS      #1,@#SR0
```

позволяет эффективно прервать связь процессора с устройствами ввода-вывода (если только к ним уже не адресовались из другой страницы). Действительно, хотя в регистре **KPAR7** установлено правильное смещение для седьмой страницы, сбрасывание битов командой **BIC** делает страницу нерезидентной. Поскольку все регистры управления памятью расположены в одной и той же области общей шины, теперь невозможно выделить под нее другую страницу. Снова единственный выход — остановить и перезапустить процессор.

Заметьте, что замена **#6** на **#77400** в команде **BIC** дает тот же эффект. Только один квант седьмой страницы будет привязан к физической памяти, а все адреса выше **760076** окажутся недоступными.

Понятно, что под регистры ввода-вывода можно приспособить произвольно выбранную страницу. Для этого достаточно настроить ее регистры. Так как они находятся в одном и том же месте памяти, то предварительно придется либо отключить управление памятью, либо получить к ним доступ через другую страницу.

Напишем программу вывода на терминал буквы **B**, на этот раз адресуясь к буферу печати как к ячейке с виртуальным адресом **34566**. Виртуальный адрес распадается на поле номера страницы **1** и на поле смещения **14566**. Следовательно, чтобы виртуальный адрес **34566** соответствовал физическому адресу **777566**, необходимо в **KPAR1** задать базовый адрес, который определяется так:

— Физический адрес	— 777566
— Смещение	— 14566
<hr/>	<hr/>
Базовый адрес	763000

И следовательно, в **KPAR7** нужно занести значение **7630**. Вся программа, в момент входа в которую предполагается, что блок

управления памятью отключен, такова

```

                                KPAR1 = 172342
                                KPDR1 = 172302
                                SR0 = 177572
                                TPB = 34566                                ;!
START:  MOV                    #7630,@#KPAR1
                                MOV                    #77406,@#KPDR1
                                BIS                     #1,@#SR0
                                MOVB                   #102,@#TPB
                                HALT
                                .END                    START

```

УПРАЖНЕНИЯ. 1. Какое минимальное число должно быть введено этой программой в старшем байте регистра **KPDR7**?

2. Напишите программу, которая виртуальным адресам с **17000** по **20000** ставит в соответствие физические адреса с теми же номерами. Добавьте в нее запрещение на запись в ячейку **17000**.

Защита памяти. Ограничения на размеры страницы и права доступа к ней, которые можно установить в регистре описания, используются в системах с разделением времени для того, чтобы предохранить выделенные различным пользователям пространства памяти от пересечения и чтобы защитить операционную систему от всех непривилегированных программистов.

Если в команде нарушается какое-либо из этих ограничений, она выполнена не будет: блок управления памятью *игнорирует* ее. Он вызовет также программное прерывание (не взирая на приоритет ЦП) с вектором прерывания в ячейке **250**. В связи с этим надо учесть, что для *всех векторов внешних и программных прерываний* используются виртуальные адреса. Поэтому команда

```
MOV                    #SERV,@#250
```

всегда установит адрес ячейки **SERV** равным адресу программы обработки прерываний от блока управления памятью. Когда произойдет прерывание, в счетчике команд не окажется содержимого ячейки **250** общей шины. Действительно, при включенном блоке управления памятью *виртуальный* адрес **250** будет смещен, и в счетчик команд поступит содержимое ячейки, соответствующей преобразованному адресу.

УПРАЖНЕНИЯ. 1. Что происходит, если после настройки векторов прерывания программа изменяет содержимое регистра **KPAR0**?

2. Напишите программу, в заключительной части которой можно командой

MOV #SERV,@#55550

установить вектор прерывания блока управления памятью. Что произойдет, если впоследствии значение регистра **KPAR2** (но не **KPAR0**) будет изменено?

Адрес, который передается программой в качестве входной точки процедуры обработки прерываний, является виртуальным, так что он имеет отношение к блоку управления памятью. Предположим, что программа объявляет третью страницу нерезидентной, включает блок управления памятью и затем выполняет команды

MOV, #60000,@#250
TST @#60000

Первая из них корректна. Хотя в ней и фигурирует значение **60000**, никаких ссылок на содержимое ячейки с таким виртуальным адресом не делается и защита памяти не нужна. Поэтому команда благополучно установит **60000** в качестве входного адреса программы обработки прерываний.

Вторая же команда пытается обратиться к нерезидентной третьей странице. В результате блок управления памятью игнорирует ее и передает управление по виртуальному адресу **250**. Процессор загрузит содержимое виртуального адреса **60000** в счетчик команд для того, чтобы передать управление программе обработки прерываний. Это, однако, вновь приведет к срабатыванию механизма защиты памяти; снова произойдет прерывание и т. д. В итоге программа заиклится.

УПРАЖНЕНИЕ. Что случится, если программа занесет в ячейку **250** число **60000**, нулевую страницу объявит нерезидентной, включит блок управления памятью и закончит работу командой **TST @#250**?

Режимы работы процессора. Привилегированных и непривилегированных пользователей в системе с разделением времени различают *по режимам*, в которых они могут работать с процессором. В системе 11/34 имеется два режима: *оперативный* и *обычный (пользовательский)*. Сразу после включения процессор перейдет в оперативный режим и останется в нем до тех пор, пока будет выполняться системная программа или программа привилегированного пользователя. Для непривилегированного пользователя операционная система изменит режим работы ЦП, пред-

варительно предприняв шаги к тому, чтобы пользователь не смог самовольно его изменить.

Наиболее характерное отличие заключается в том, что команда **HALT** является некорректной в режиме пользователя. Вместо останова процессора она приведет к прерыванию в ячейке 4 или 10. (В какой именно, зависит от процессора; руководства в данном случае помочь не могут, поэтому лучше всего проверить непосредственно. Было обнаружено даже, что в операционной системе RSX-11 печатается сообщение «сбой системы в ячейке 4», тогда как на самом деле на процессоре 11/34 управление передавалось в десятую ячейку.)

Режим задается 14-м и 15-м битами слова состояния процессора **PS**. Слово состояния имеет на общей шине адрес 777776. Если оба бита равны 0, процессор работает в оперативном режиме. Если же это единицы, то режим обычный. На процессоре 11/34 не допускается, чтобы значения битов были различны, однако на более мощных машинах серии PDP-11 такой запас используется для введения большего числа режимов.

На процессоре 11/45 и других, более мощных моделях программе, исполняющейся в пользовательском режиме, аппаратно запрещено изменять значения указанных битов **PS**. Но на модели 11/34 единственный способ запрещения установки оперативного режима в программах, выполняющихся в режиме пользователя, заключается в использовании механизма защиты памяти. Это позволяет сделать регистр **PS** недоступным программам непривилегированных пользователей, так как на самом деле существует два полных набора регистров адресов и описаний страницы: один для оперативного, а другой для обычного режима.

До сих пор наше изложение, в том числе касающееся адресов регистров на общей шине, относилось к управлению памятью в оперативном режиме. В обычном режиме оно, по существу, аналогично. Адреса общей шины такие:

Номер страницы	UPAR	UPDR
0	777640	777600
1	777642	777602
2	777644	777604
3	777646	777606
4	777650	777610
5	777652	777612
6	777654	777614
7	777656	777616

Когда блок управления памятью получает виртуальный адрес, то он прежде всего анализирует слово состояния, чтобы определить, в каком режиме находится процессор. В зависимости от режима в описанном ранее алгоритме вычисления смещения

используется тот или иной набор регистров. Содержимое этих наборов в целом не зависит одно от другого. Таким образом, допустимо (это и практикуется), что в пользовательском режиме программам будет закрыт доступ к регистрам ввода-вывода.

Тем не менее выполняющейся в обычном режиме программе должны быть предоставлены возможности для осуществления ввода-вывода. Мы уже знаем, что это достигается при помощи системных подпрограмм, и сейчас можем более детально ознакомиться с тем, как это происходит. Давайте в последний раз напишем программу печати на терминале буквы В. Мы достигнем цели командой **TRAP** в обычном режиме. Считая, что перед входом в программу процессор находится в оперативном режиме, установим программное прерывание:

```
MOV      #TRP,@#34
MOV      #340,@#36
```

Метку **TRP** мы выбрали в качестве входной метки процедуры обработки программных прерываний и установили приоритет последней равным 7, так что можно не беспокоиться по поводу прерываний от внешних устройств. Заметьте, что значения адресов программных и внешних прерываний определены в виртуальном адресном пространстве *оперативного* режима. Они будут доступны программе в другом режиме, если только какая-то страница виртуального адресного пространства обычного режима соответствует тем же самым физическим ячейкам. Обычно же блок управления памятью накладывает запрет на доступ к этим векторам в пользовательском режиме.

В ячейке с меткой **TRP** имеем

```
TRP:      MOVB      #102,@#TPB
          RTI
```

Естественно, что идентификатор **TPB** должен быть где-то описан. Предполагая, что регистр **KPAR7** настраивается, как обычно, полагаем **TPB=177566**.

Метке **TRP** загрузчик поставит в соответствие некий виртуальный адрес. При помощи блока управления памятью можно убедиться в том, что соответствующая ему ячейка общей шины недоступна программам в обычном режиме. В крайнем случае ее можно установить только на чтение. Тогда программа обычного режима могла бы выполнить команду **JMP TRP**, не вызвав обращения к блоку защиты памяти. Все же, поскольку регистры устройств ввода-вывода недоступны программам обычных пользователей, команда с меткой **TRP**, по-видимому, не выполнится.

УПРАЖНЕНИЕ. Завершите программу, включив в нее настройку регистров управления памятью для защиты векторов

прерывания, регистров устройств и системных процедур от программ непривилегированных пользователей.

Теперь можно установить пользовательский режим:

BIS #140000,@#PS

При этом считаем, что **PS** указывает на ячейку **777776** общей шины. Теперь все попытки непосредственно выполнить операции ввода-вывода вызовут программное прерывание в ячейке **250** оперативного виртуального адресного пространства. Мы не будем писать саму программу обработки прерывания, но по крайней мере должны проконтролировать его, для чего в ячейку **250** занесем код **252**, а в ячейку **252** — нуль. (Что при этом произойдет?)

Однако можно воспользоваться и командой **TRAP**. Она заносит содержимое регистров **PS** и **PC** в стек и загружает в **PC** слово из ячейки **34**, а в **PS** из ячейки **36**, причем адреса рассматриваются в оперативном виртуальном адресном пространстве. Поскольку 14-й и 15-й биты ячейки **36** нулевые, *новая загрузка PS приведет к установке оперативного режима*. Следовательно, исполнится команда **MOVB** с меткой **TRP**, и буква **B** появится на терминале. Команда **RTI** восстановит первоначальные значения **PC** и **PS**, *возвращая систему к обычному режиму*.

УПРАЖНЕНИЕ. Напишите программу полностью.

На практике было бы неудобно, если бы операционная система с разделением времени отвечала на все запросы пользователя по выводу обращением к терминальному буферу печати с адресом **777566**, потому что он относится к *терминалу оператора*, который расположен рядом с ЦП в машинном зале. Реально связь между ЦП и различными пользовательскими терминалами осуществляется посредством адресации к регистрам сложного интерфейсного устройства.

Связь между адресными пространствами. В § 3.4 было продемонстрировано, как системная программа **ЕМТ** может выбрать параметр из младшего байта команды **ЕМТ**. Предположим теперь, что обращение к команде **ЕМТ** происходит, когда процессор находится в обычном режиме. Программа **ЕМТ** сохраняет значения регистров от **R0** до **R5** в стеке, и поэтому адресом возврата оказывается **14(SP)**. Он загружается в **R0**:

MOV 14(SP),R0

Теперь сама команда **ЕМТ** расположена по виртуальному адресу **—2(R0)**, но в виртуальном пространстве *обычного* режима. Про-

грамма **ЕМТ**, однако, будет функционировать в оперативном режиме. Допустим, что она была загружена с виртуального адреса **2000** обычного пространства. Если она заканчивается командой

MOV **—(R0), —(SP)**

то в стек запишется содержимое виртуального адреса **2000** пространства *оперативного* режима. Это будет та же самая физическая ячейка памяти, если только нулевая страница в обоих режимах одинаково распределена.

Вместо предыдущего способа в системном вызове нужно использовать команду засылки из пространства предыдущей команды **MFPI** (Move From Previous Instruction space). Это одноадресная команда. Ее операнд интерпретируется как результат вычисления исполнительного адреса, соответствующего виртуальному адресу предшествующего пространства (в нашем случае — пользовательского). Команда заносит содержимое этого адреса в системный стек. Поэтому программа **ЕМТ** должна заканчиваться так:

MFPI **—(R0)**

Обратная связь — от стека к адресу предшествующего пространства — достигается аналогичным образом при помощи команды записи в пространство предшествующей команды **МТРІ** (Move To Previous Instruction space).

Предшествующий режим определяется как режим, в котором процессор находился перед последним программным или внешним прерыванием. Причем, когда одно из них происходит, в слово состояния процессора заносятся значения битов второго слова вектора прерывания, за исключением битов 12 и 13: их состояние определяется предшествующими значениями битов 14 и 15 в **PS**. В командах **MFPI** и **МТРІ** осуществляется проверка 12-го и 13-го битов **PS**, позволяющая определить, о каком виртуальном адресном пространстве идет речь.

УПРАЖНЕНИЯ. 1. Используя команду **TRAP 'X**, дополните рассмотренную нами программу так, чтобы на терминале можно было напечатать любую букву.

2 *. Напишите свои собственные макро, эквивалентные по действию командам **MFPI** и **МТРІ**.

Системные стеки. В каждом из режимов процессор заводит особый указатель стека. Указатели представляют собой совершенно различные регистры процессора, которые устанавливаются

ся независимо один от другого. В то же время программа сама по себе не может предпочесть один стек другому: выбор предопределен режимом, в котором находится процессор.

Поэтому, если в программе определено выражение $SP = \%6$, то любая ссылка на идентификатор **SP** (Stack Pointer) будет интерпретироваться аппаратурой как обращение к стеку оперативного режима, если процессор находится в оперативном режиме, и как обращение к стеку обычного режима, если ЦП находится в этом режиме. В подавляющем большинстве случаев для пользователя не имеет значения, какой указатель использует операционная система.

Когда происходит программное или внешнее прерывание, ЦП анализирует 14-й и 15-й биты второго слова вектора прерывания, чтобы определить, в каком стеке хранить значения **PC** и **PS** для возврата в вызывающую программу. Такая проверка обеспечивает корректный возврат по команде **RTI**. Ясно, что программа обработки прерывания не должна перед выходом изменять режим процессора, потому что тогда команда **RTI** обратится к другому стеку. Стоит отметить, что командами **MFPI** и **MTPI** осуществляется обмен информацией между предшествующим пространством и стеком, соответствующим режиму исполнения текущей команды.

Адреса параметров некоторых системных вызовов должны загружаться в стек до адресации. Процессу, протекающему в оперативном режиме, требуются две команды для выбора параметра из стека пользователя. Во-первых, в стек оперативного режима загружается значение указателя стека пользователя, которое и есть адрес передаваемого параметра:

MFPI **SP**

Обратите внимание на то, что, если в командах **MFPI** и **MTPI** используется режим *регистровой адресации* (режим 0), процессор интерпретирует это как ссылку на стек предшествующего режима.

Так как адрес аргумента теперь находится в оперативном стеке, ссылка на сам параметр имеет вид **@(SP)**, т. е. через *оперативный* стек, но в виртуальном пространстве *обычного* режима. Чтобы поменять местами параметр и его адрес, достаточно записать

MFPI **@(SP) +**

используя тот же прием, что и для передачи управления между сопрограммами. Важно заметить, что, когда в командах **MFPI** и **MTPI** режим адресации к **SP** отличен от регистрового, процес-

сор интерпретирует его как ссылку на стек текущего режима, а полученный исполнительный адрес относит к предшествующему пространству.

УПРАЖНЕНИЯ. 1*. Напишите последовательность команд, позволяющую занести значение параметра в стек пользователя для программы, выполняющейся в оперативном режиме.

2*. Исправьте свои макро **MFPI** и **MTPI** так, чтобы в них учитывалось наличие двух стеков.

ПРИЛОЖЕНИЕ А. ODT

ODT — это комплекс отладки в режиме on line (On line Debugging Technique), входящий в состав операционных систем DEC PDP-11. Его реализация, как и реализация любой другой системной программы, зависит от принятой версии операционной системы. Однако основные возможности ODT остаются в большей мере неизменными, несмотря на различия в формате команд.

Работа ODT

Как правило, программа ODT поставляется в виде перемещаемого двоичного файла **ODT.OBJ**. Он должен быть скомпонован с отлаживаемой программой. Допустим, что нам нужно отладить программу **TEST.MAC** посредством ODT. Сначала ее надо оттранслировать, а потом при помощи компоновщика *скомпоновать* перемещаемый модуль **TEST.OBJ** с ODT:

```
TEST,TEST=ODT,TEST↵
```

Эта команда указывает компоновщику на два входных файла. Он связывает их друг с другом и создает два выходных файла: **TEST.SAV** и **TEST.MAP**. Имейте в виду, что оба эти файла представляют собой снимок оперативной памяти программы **TEST** вместе с ODT. Поэтому, если нужно сохранить образ памяти с одной лишь вашей программой, то при компоновке вы можете просто задать различные имена для результирующих файлов.

В зависимости от компоновщика может оказаться необходимым, чтобы в списке компонуемых файлов файл с ODT был расположен первым, как это и показано выше. Причина этого заключается в следующем: после того как системная команда

```
RUN TEST↵
```

загрузит файлы в память, она обязательно осуществит переход на начальный адрес программы отладки, а не на вход программы **TEST**. Когда это произойдет, отладчик напечатает краткое сооб-

щение о номере своей версии и затем символ *, означающий, что он готов принимать указания.

Прежде чем идти дальше, необходимо научиться легко выполнять описанный процесс на имеющейся в вашем распоряжении машине. Обычный способ выхода из ODT на уровень монитора состоит в наборе на пульте ^C в момент, когда отладчик ожидает очередную команду.

Глобальные имена

Один из способов использования комплекса ODT заключается в прослеживании на различных стадиях вычислительного процесса изменений содержимого некоторых особенно важных ячеек. В команды отладчику должны при этом входить не имена ячеек, присвоенные пользователем, а значения их адресов. Так, мы не можем заглянуть в ячейку с меткой **MEM**, пока нам не известно, куда она была загружена. В файле **.MAP** не содержится подобной информации; по нему даже нельзя определить начальный адрес отлаживаемой программы, поскольку она теперь лишь часть программного комплекса, точка входа в который совпадает с адресом запуска отладчика.

Можно, однако, побудить компоновщик при создании файла **.MAP** указать начальный адрес нашей программы. Это достигается путем объявления имени *глобальным* (т. е. доступным вне программы) с помощью директивы

```

                                .GLOBL    START
                                ...
START:                        ...

```

Такая директива необходима также при независимой трансляции различных подпрограмм и затем последующего объединения их компоновщиком. Предположим, что нам нужно скомпоновать два файла **MAIN.MAC** и **SUB.MAC** после того, как они будут оттранслированы, и что файл **SUB.MAC** содержит подпрограмму **SUBRTN**. Если в первом файле стоит вызов подпрограммы, то они *оба* должны содержать описание **.GLOBL SUBRTN**.

Отдельно транслируемые части программы или *модули* должны оканчиваться директивой **.END**. Причем начальный адрес всей программы должен быть задан в качестве параметра этой директивы только того модуля, в котором он определяется. Все же остальные модули завершаются просто **.END**. При компоновке с отладчиком чаще всего нет необходимости выкидывать из отлаживаемой программы адрес запуска, так как в большинстве операционных систем предусмотрено, чтобы управление сразу передавалось комплексу ODT.

Отладка и исправление программы

Перед тем как компоновать программу в ODT, нужно объявить ее адрес запуска глобальным. Кроме того, *следует вооружиться листингом своей программы* (без ODT).

После компоновки отлаживаемой программы в ODT выйдите из компоновщика и посмотрите в файл **.MAP**. Запомните значение начального адреса по карте загрузки. Потом запустите сформированную программу, чтобы в результате управление перешло к отладчику и он напечатал символ *.

Допустим, к примеру, что по листингу ячейка **МЕМ** имеет адрес **162**. Его значение, конечно, берется относительно начала программы. Поэтому если в файле **.MAP** указано, что начальный адрес загрузки равен **6316**, то при работе с ODT к ячейке **МЕМ** можно адресоваться по **6500 (=6316+162)**.

Так как утомительно каждый раз вручную выполнять подобные расчеты, то первой инструкцией отладчику необходимо задать базовый адрес, относительно которого рассматриваются все последующие адреса. В ODT имеется набор из восьми *регистров смещения*, пронумерованных от 0 до 7. Это позволяет устанавливать базовые адреса не менее чем восьми независимым программным модулям. Нам же нужно задать только одну константу (**6316**) для всей программы, поэтому в ответ на приглашение отладчика набираем **6316;R** (без **↵** в конце), что приводит к занесению числа 6316 в регистр 0. Отладчик ответит новым приглашением *. В дальнейшем при наборе на терминале любого адреса будем ставить перед ним 0, отмечая тем самым, что он берется относительно содержимого нулевого регистра. Чтобы передать требуемый адрес отладчику, всегда нужно указывать номер используемого вами регистра, как видно из приведенных ниже примеров.

Теперь можно передавать отладчику в точности те адреса, которые имеются в листинге трансляции, что очень удобно. Пусть по относительному адресу **72** в нашей программе стоит команда **CLR R1**. Если мы наберем **0,72/(здесь также не нужен ↵)**, ODT на той же строке ответит: **005001**. Символ / указывает ODT на слово, адрес которого был только что набран, и требует, чтобы он выдал его содержимое. Важно понимать, что, прежде чем сделать это, отладчик выполнит две операции: он сместит указатель текущей ячейки или *курсор* в относительный адрес **72** и *откроет* слово по этому адресу.

Если ячейка была хоть раз открыта, ее содержимое можно изменить. Предположим, что команда в нашем примере, как оказалось, должна быть другой: **CLR @R1**. Тогда после вывода отладчиком кода **005001** нужно набрать новое значение, т. е. **005011**, и на этот раз закончить строку возвратом каретки **↵**. Символ

←| сообщает отладчику о том, что нужно закрыть слово, записав в него новое содержимое. Если же изменять ничего не требуется, то просто нажмите ←|. Поскольку нового содержимого задано не было, ODT закроет слово, оставив в нем все по-старому.

Те исправления, которые вы внесли в программу с помощью отладчика, не сохранятся в файле: они будут утеряны, как только вы выйдете из программы ODT. Поэтому их нужно фиксировать в копии листинга вашей программы в процессе отладки.

Если вы ошиблись во время набора очередной команды, то клавишей RUBOUT нельзя стереть неверный символ. Нажатие этой клавиши не противозаконно, но оно приведет к игнорированию отладчиком всей команды и появлению символа *. В результате придется ввести всю команду повторно.

При закрытии ячейки курсор отладчика не смещается. Нажатие клавиши / приводит к открытию ячейки, на которую в данный момент указывает курсор, и выводу на пульт ее содержимого. Это удобно для проверки правильности только что внесенных изменений.

После изменения содержимого ячейки (если оно было необходимо) ее можно закрыть не только возвратом каретки ←|, но также вводом любого из символов: ↓ (LINE FEED), ^, ←, @, > или <. На некоторых терминалах литере ← соответствует литера «подчеркивание».

При построочном просмотре программы наиболее целесообразным представляется использование клавиши LINE FEED, так как ее нажатие приводит не только к закрытию текущей ячейки, но и к сдвигу курсора на следующую, открытию ее и выводу на пульт ее содержимого. Пусть, например, после команды **CLR R1**, расположенной по адресу 72, следует команда **MOV R3, MEM**. Закрытие ячейки 72 при помощи ↓ вместо ←| приведет к появлению на терминале строки 0,000074/010367. Она указывает, что текущим адресом будет 74 с учетом содержимого нулевого регистра. Ячейка 74 теперь открыта, и по желанию можно либо изменить ее, либо просто закрыть.

Закрытие ячейки клавишей ^ аналогично ↓, но движение происходит в противоположном направлении: указатель сдвигается к предыдущему адресу, открывается соответствующая ячейка и на терминал выдается ее содержимое.

Если метка **MEM** имеет относительный адрес 162, то закрытие ячейки 74 клавишей ↓ приведет к 0,000076/000062. По листингу программы легко удостовериться, что это правильная ссылка на **MEM** при относительной адресации. Расчет смещения выполняется в предположении, что счетчик команд указывает на следующую ячейку. Поэтому относительный адрес ячейки памяти, на которую ссылается команда, такой: $(72+2)+62=162$.

Заккрытие ячейки клавишей ← приводит к индексированию содержимого текущего слова по отношению к адресу следующей ячейки: как мы только что видели, в результате получается адрес слова в относительной адресации. Отладчик открывает слово с таким адресом и выводит на терминал его содержимое. Так, в нашем случае в результате закрытия ячейки 76 клавишей ← будет напечатано 0,000162/000000 в предположении, что при трансляции в ячейку MEM был занесен нуль. Таким образом, клавиша ← позволяет проследить эффект от выполнения подобных команд.

Учтите, что не существует специальной команды, открывающей ячейку с адресом, который вычисляется с помощью индексирования по какому-либо иному регистру, отличному от PC. Команда ← сама по себе не использует значение счетчика команд, потому что он указывал бы на подпрограмму в комплексе ODT, отвечающую за выполнение этой команды. Зато в ней имеется специальный указатель, с помощью которого определяется, к какой ячейке во время исполнения будет произведено обращение при относительной адресации.

Если бы вместо рассмотренной выше команды у нас была бы **MOV R3, @#MEM**, то открытие ячейки 76 дало бы абсолютный адрес метки MEM: 0,000076/006500. Чтобы заглянуть на этот раз в ячейку MEM, нужно закрыть текущую ячейку нажатием клавиши @. В ответ отладчик выведет 0,000162/000000, интерпретируя содержимое закрываемого слова как адрес ячейки, которую нужно открыть. Поскольку мы еще не исполняли программу и ничего не изменяли в ней, ячейка MEM, как и следовало ожидать, содержит в точности то, что было занесено в нее директивой **.WORD** во время трансляции.

Если до применения команд ← или @ изменить содержимое закрываемого слова, то именно оно будет использовано для вычисления адреса следующей открываемой ячейки.

Для вызова командами **JSR** или **JMP** других подпрограмм можно применить эти же команды в зависимости от способа адресации к месту передачи управления. Поскольку открываемая ячейка есть начало соответствующей подпрограммы, представление последней в машинной памяти можно просмотреть клавишей ↓. Для проверки же команд перехода нужно использовать команду >. Допустим, из листинга программы следует, что по адресу 200 стоит команда **BNE MEM**. Если метка MEM имеет адрес 162, то отладчик выдал бы 0,000200/001370, так как условный переход должен произойти десятью словами раньше относительно ячейки, следующей за командой перехода. Если мы теперь командой > закроем открытую ячейку, то ODT выведет на терминал 0,000162/000000. Ячейка, в которую делается переход, вычисляется отладчиком по значению младшего байта зак-

рываемого слова. Отладчик не проверяет, действительно ли в старшем байте закодирована команда перехода. Если перед обращением к команде $>$ содержимое текущего слова было изменено, то ячейка, на которую произойдет переход, будет вычислена по новому значению.

В процессе отладки вы можете обнаружить, что ссылка на ячейку с относительным адресом приводит к неправильному адресу. Допустим, что команду в ячейке 200 нужно заменить на **BNE LABEL**, где **LABEL** имеет адрес 316. Сначала открываем ячейку 200 командой **0,200/**, в результате чего ODT выведет на терминал **001370**. Теперь командой **;0** даем указание отладчику рассчитать смещение для перехода к ячейке 316. На терминале набираем **0,316;0**. Отладчик ответит строкой **000114 046**, выдавая значение смещения в байтах для команд с относительной адресацией и в словах для команд перехода. Если в последнем случае оно выходит за диапазон, допустимый в командах перехода, то напечатано не будет.

Нам нужно изменить команду перехода, поэтому набираем **001046←|**, что приводит к нужному результату и закрывает слово. Заметьте, что набор на терминале просто **46←|** не сработает (хотя в то же время команда **1046←|** корректна).

Команды \leftarrow , $@$ и $>$ нарушают последовательный просмотр программы. Однако в программе отладчика сохранится запись содержимого слова, которое было раскрыто последним перед командой, нарушающей обычный порядок. Команда $<$ вернет указатель к соответствующему адресу и в дальнейшем будет эквивалентна \rightarrow . Пусть в программе по относительному адресу 220 находится команда **JSR PC,PRINT**. Тогда ссылка по относительному адресу **PRINT** расположена в ячейке 222, и, чтобы просмотреть вызываемую подпрограмму, нужно закрыть указанную ячейку командой \leftarrow . Затем используем \downarrow для последовательного просмотра ячеек подпрограммы. Когда мы просмотрим достаточно ячеек, закрываем текущее слово командой $<$. В результате открывается ячейка 224, т. е. следующая за последней перед переходом к подпрограмме **PRINT** ячейка, и будет выведено ее содержимое. Но повторное нажатие клавиши $<$ не приведет к возвращению еще на один уровень назад, а будет равносильно команде \downarrow .

Чтобы просмотреть содержимое регистров, нужно в команде $/$ после символа $\$$ набрать номер регистра. Так, после ввода команды **$\$3/$** отладчик распечатает содержимое регистра **R3**. Его можно заменить новым значением, которое вводится обычным способом перед закрытием регистра. При закрытии регистра клавишами \downarrow или \wedge открывается соответственно следующий или предыдущий регистр. Можно также применять команды \leftarrow или $@$, но не $>$ и $<$. Такое ограничение естественно, по-

скольку программа не может передавать управление регистру. Следовательно, он не может содержать команду перехода, и в то же время применение команд \leftarrow , @ или $>$ никогда не приведет к его открытию.

Внутренний регистр отладчика \$S содержит коды состояния процессора (в обычном порядке) в виде четырех младших битов слова состояния. Их можно посмотреть, набрав на терминале \$3/. На другие разряды регистра \$S обращать внимание не следует.

Комплекс ODT позволяет также выводить содержимое памяти в виде литер в коде ASCII или 50-ричном коде. Инструкция \ применяется для открытия байта, распечатки его содержимого в числовом виде и затем, если это возможно, в виде литер в коде ASCII. Так, если в программе меткой MEM помечена директива

MEM: .WORD "AB

то команда 0,162/ выдаст на терминал 041101, в то время как, набрав 0,162\, в результате получим 101=A. Теперь отладчик перешел в байтовый режим, и команды ↓ и ^ будут обращаться к соответствующим байтам. Например, после нажатия клавиши ↓ отладчик выведет информацию:

0,000163 \102 =B

Команда / с четным адресом возвратит ODT в режим работы со словами. Применение же ее с нечетным адресом эквивалентно команде \.

Для проверки текста в 50-ричном коде используйте, как и обычно, клавишу /, чтобы выдать содержимое нужного вам слова в восьмеричном формате. После этого наберите литеру X (без ←|), в результате чего отладчик выдаст содержимое слова в виде трех литер 50-ричного кода. При закрытии ячейки применяйте только команды ←| или ↓, так как в некоторых версиях программы ODT иные закрывающие ячейку команды будут интерпретироваться (безуспешно) как новое содержимое (в 50-ричном коде) текущей ячейки.

Выполнение программы

Команда ;G сообщает отладчику о том, что нужно передать управление по указанному адресу. Поэтому, если набрать 0,0;G, программа будет выполняться с начального адреса, если только базовый регистр отладчика был правильно установлен, как было показано выше. Однако, как правило, не имеет смысла применять рассматриваемую команду без некоторых предварительных дейст-

вий, потому что программа просто будет исполнена, как если бы ее запустили без всякого отладчика.

Простой путь эффективного использования команды ;G заключается в проверке работоспособности программы, начиная с некоторой точки. Допустим для иллюстрации, что нам нужно проверить правильность работы программы вывода. Если она начинается с относительного адреса 330, то после занесения в соответствующее место памяти тестовых данных в ответ на приглашение отладчика * вводим 0,330;G. Все же и это не самый эффективный путь использования возможностей ODT, потому что после каждого запуска программа будет «вылетать». Перед запуском с определенного адреса тестируемой программы вы должны быть уверены, что в определенном месте исполнение будет приостановлено. Это достигается включением в программу *точек останова*. Сначала мы обсудим, как заводить и перемещать точки останова, а потом как ими пользоваться.

Чтобы установить точку останова в заданной ячейке памяти, нужно набрать ее адрес и за ним две литеры: ;B. Отладчик ответит новым приглашением *. Таким путем можно завести до восьми точек с номерами от 0 до 7. Причем, если мы специально не указали номер текущей точки останова, ей автоматически присваивается наименьший из незанятых номеров. К примеру, команда 0,620;6B установит точку останова номер 6 в ячейке 620 программы, если только этот номер уже не был присвоен какой-либо другой точке. Если же он был задан, то его можно убрать, для чего в ответ на приглашение * отладчика нужно набрать команду ;6B, после которой данный номер освобождается. Обратите внимание, что в последней команде не указывается текущий адрес точки останова. Чтобы убрать все точки останова, надо набрать просто ;B.

Если вы забыли, куда поставили какую-то конкретную точку, то наберите \$B/, в ответ на что отладчик выдаст адрес точки с нулевым номером. Далее, применяя ↓, последовательно просмотрите адреса остальных точек.

Заводить или изменять точки останова нужно до запуска программы командой ;G либо после завершения отладчиком выполнения одной из команд ;P (о которых будет сказано ниже) и выхода его на символ *.

Необходимо упомянуть об одном ограничении. Подпрограмма комплекса ODT пересылает содержимое ячеек, в которых устанавливаются точки останова, командой BPT (BreakPoint Trap) и восстанавливает его, когда точки останова удаляются. Следовательно, точку останова нельзя установить в ячейке, на содержимое которой ссылается другая программа. Так, например, если должна выполняться команда JSR PC,@TABLE, то в ячейку TABLE точку останова ставить нельзя. Если же

команда имеет вид **JSR PC, TABLE**, то никакого вреда от точки останова не будет.

Начнем с заведения единственной точки останова по начальному адресу программы, набрав в соответствии со сказанным директиву **0,0;B**. После этого запускаем программу директивой **0,0;G**. Исполнение команд программы продолжается до тех пор, пока не будет достигнута точка останова, и перед выполнением той команды, на месте которой она стоит, управление передается отладчику.

Поскольку в нашем случае точка останова находится в самом начале, отладчик остановится до выполнения каких бы то ни было команд программы и сообщит нам о том, что происходит, строкой на терминале: **B0;0,000000**, открывая адрес, по которому прервано исполнение; после этого он будет ждать дальнейших инструкций. Теперь мы можем отлаживать программу покомандно. Прежде всего устанавливаем *пошаговый режим* командой **;1S**. В действительности здесь вместо единицы можно использовать любую цифру, отличную от нуля, однако эту команду надо как-то отличить от аналогичной по написанию команды **;S**, о которой речь впереди.

Как только пошаговый режим установлен, инструкцией **;P** заставляем отладчик выполнить следующую команду. Если наша программа начинается двухадресной командой, то она будет выполнена и ODT выдаст **B8;0,000004**, выводя адрес следующей подлежащей исполнению команды. Если только что исполненная команда была командой перехода или условного перехода, условие которого оказалось выполненным, то отладчик выдает адрес точки перехода, как оно и должно быть, так как это адрес следующей исполняемой команды. Обратите внимание, что в пошаговом режиме выдается несуществующая, восьмая точка останова.

На данном этапе отладки можно посмотреть и изменить содержимое любых ячеек памяти, потом инструкцией **;P** вызвать исполнение следующей команды и т. д. В любой момент вместо **;P** можно набрать **;S**, чтобы сбросить пошаговый режим исполнения и далее директивой **0,0;G** запустить программу сначала. Установленная нами точка останова сохранится, и все сделанные исправления останутся в силе.

Когда по инструкции **;P** адресуются к **EMT** или **TRAP**, то не только они, но и вся процедура прерывания, а также еще одна команда основной программы немедленно исполняются. По причинам, которые вам станут ясны, если вы прочитаете гл. 4, существуют трудности в пошаговом исполнении мониторной программы ввода. Лучше ставить останов сразу после обращения к ней.

Если ввести инструкцию ***n*;P**, где ***n*** — восьмеричное число, то отладчик пропустит ***n*** команд программы, прежде чем вернет себе управление.

Другой способ более быстрого продвижения по командам заключается в применении инструкции **;P** без введения пошагового режима исполнения (или после отмены его командой **;S**). В результате исполнение начнется с последней невыполненной команды и будет продолжаться до тех пор, пока не встретится очередная точка останова. Снова перед выполнением команды в этой точке управление перейдет к отладчику. В промежутке между этими событиями никакая информация выдаваться не будет (если только в самой программе нет операций вывода и не возникнут ошибки, на которые специальная программа отреагирует сообщением об ошибке).

Если внутри отлаживаемого отрезка требуется ввод с терминала, пользователь должен набрать все, что нужно. Будьте внимательны и не вводите данные, когда отладчик ожидает от вас очередную инструкцию, и наоборот.

Для каждой из восьми точек останова отладчик заводит *счетчик прохождений*. Если точка останова инструкцией **адрес;В** заводится впервые, то ее счетчик сбрасывается.

Когда в процессе исполнения под управлением ODT точка останова достигается, отладчик проверяет содержимое счетчика. Он уменьшает его на единицу и проверяет, будет ли результат меньше или равен нулю. Если да, то отладчик заносит в счетчик 1, выдает сообщение о попадании в данную точку и ждет дальнейших указаний. Если же результат строго положителен, то он возвращает управление программе пользователя. Поэтому для того, чтобы отладчик ***k*** раз пропустил заданную точку останова, в ее счетчик нужно заслать число ***k***. Если программа отладки вышла в некоторую точку останова, то командой ***k*;P**, где ***k*** — восьмеричное число, устанавливается значение счетчика этой точки равным ***k***, а дальнейшее ее действие эквивалентно команде **;P**. Заметьте, что команды **1;P** и **;P** тождественны.

Можно устанавливать счетчик прохождений, даже если программа не была остановлена в соответствующей точке. Как было сказано, внутри программы ODT по адресу **\$B** расположен блок из восьми ячеек, в которых хранятся адреса точек останова. Сразу после этого блока находится ячейка, в которую заносится адрес следующей исполняемой команды в пошаговом режиме работы. А после нее идет блок, содержащий счетчики точек останова. Их значения можно установить командами отладчика. Надо только не ошибиться в подсчете числа нажатий клавиши ↓ после того, как была раскрыта ячейка **\$B**.

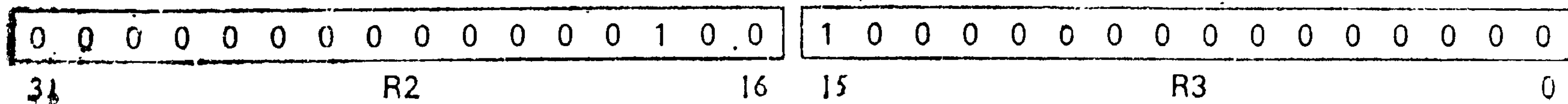
В предыдущем изложении мы рассматривали основные арифметические операции над целыми числами в предположении, что результат всегда помещается в одно слово машины PDP-11. Теперь обсудим возможности ассемблера MACRO-11 при работе с большими числами, не обязательно целыми. Здесь мы не можем позволить себе ничего иного, кроме беглого просмотра обширного материала, полное понимание которого требует достаточно высокого уровня математической подготовки. Цель наша состоит в том, чтобы арифметические команды макроассемблера были понятны студенту, который уже изучил или изучит основы вычислительной математики. В этом приложении будет также предполагаться более глубокое знакомство с двоичной и восьмеричной арифметикой, чем в остальных частях книги.

Команды MUL и DIV

Приведенное ранее описание этих команд дает о них упрощенное представление. Если в команде **MUL MEM, R** используется нечетный регистр **R**, то все верно. Если же используется четный, то произведение в виде тридцатидвухразрядного числа заносится в регистр **R** и регистр со следующим номером. Рассмотрим такой пример:

```
MOV      #1100,R2
MUL      #1000,R2
```

Результат, если его рассматривать как тридцатидвухразрядное число, содержит 1 в битах 15 и 18 и нули в остальных. Младшие шестнадцать битов запоминаются в R3, а старшие — в R2. Таким образом, в данном случае после выполнения команды MUL третий регистр содержит число 100000, а второй 4:



В команде **MUL** операнды считаются целыми двоичными числами со знаком, а результат формируется в виде тридцатидвухразрядного числа, 15-й бит старшего слова которого является знако-

вым. Нужно заметить, что результат умножения всегда корректен, потому что произведение двух шестнадцатиразрядных чисел не выходит за пределы двойного слова.

Команда **DIV** выполняет деление целого числа в форме двойного слова, расположенного в регистре с четным номером и следующим за ним:

```
MOV      #1,R0
MOV      #100007,R1
DIV      #400,R0
```

В этом примере две первые команды загружают число 300007 в регистры **R0** и **R1**. Команда **DIV** частное (600) запишет в **R0**, а остаток (7) — в **R1**.

Если бы первой шла команда **MOV #1000, R0**, то результат от деления не поместился бы в одно слово. Действительно, при делении числа 200100007 на 400 получается 400200. При подобных попытках выполнение команды прерывается и устанавливается бит **V**.

УПРАЖНЕНИЕ. Изучите работу команды деления, когда один или оба операнда отрицательны; особое внимание при этом обратите на остаток.

Целая арифметика повышенной точности

Даже самые незамысловатые вычисления могут привести к результатам, которые слишком велики, чтобы уместиться в шестнадцать битов машинного слова **PDP-11**. Поэтому довольно часто оказывается необходимым отводить дополнительные ячейки для записи одного числа. В большинстве случаев бывает достаточно двух слов (*двойная точность*), однако ненамного сложнее рассмотреть и более общий случай.

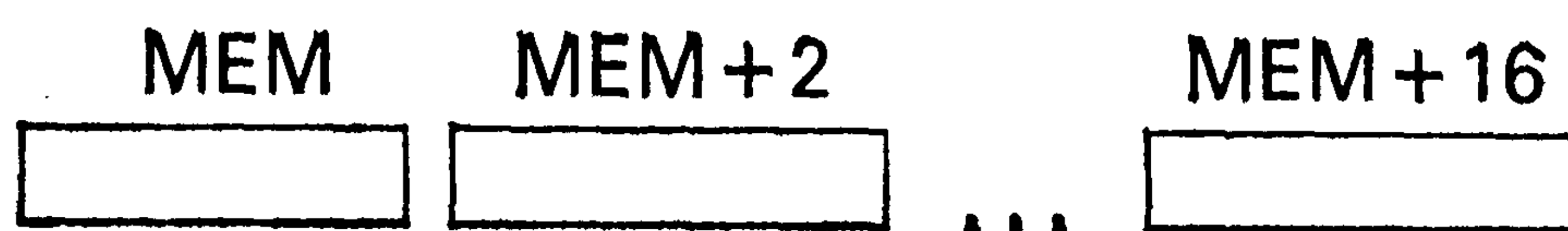
Предположим, нам нужно научиться оперировать числами, имеющими до 35 десятичных знаков включительно. Их можно рассматривать как значащие цифры числа в представлении с плавающей десятичной точкой, и в таком случае точность в тридцать пять знаков после запятой безусловно выглядит чрезмерной для большинства прикладных задач. Так как $\log_{10} 2 \approx 0.3$, то 2^{117} имеет тот же порядок, что и 10^{35} . Следовательно, для представления чисел в рассматриваемом диапазоне нам требуется около 117 битов. Поскольку восемь машинных слов содержат 128 битов, то это и необходимо, и достаточно.

Представление чисел с повышенной точностью аналогично представлению числа в одном слове — просто увеличивается количество разрядов, отводимых под число. Вообразите себе число, размещенное в восьми ячейках блока, начинающегося с **МЕМ**

и оканчивающегося в $MEM+16$ (проверьте правильность конечного адреса!). Если это 1, то все биты слов от MEM до $MEM+14$, будут равны нулю, а в ячейке $MEM+16$ нулевой бит будет равен 1, а остальные — нулю. Если число равно 400000, то только семнадцатый бит его отличен от нуля, т. е. первый бит ячейки $MEM+14$.

Отрицательные числа будут представлены в виде 128-разрядного двоичного дополнения. Так, числу -1 соответствует 1 во всех битах блока.

Нет никакой существенной причины, почему ячейки с последовательно возрастающими адресами должны содержать убывающие по значимости разряды числа. Такое соглашение принято, скорее, интуитивно, поскольку в приведенном ниже изображении блока



и возрастание адресов его элементов, и убывание значимости разрядов числа оказываются расположенными в их естественном порядке: слева направо. Оно также согласуется с особенностями специальных команд некоторых процессоров серии PDP-11, которые выполняют арифметические действия над числами с двойной точностью в предположении, что в слове с меньшим адресом хранятся старшие биты. Поэтому такого соглашения желательно придерживаться.

Каждая «переменная» повышенной точности будет представлена в нашей программе идентификатором, указывающим на начало блока из восьми слов, в котором хранятся ее «цифры». Допустим, что в программе должно быть заведено десять таких чисел. Под них нужно отвести достаточное место в памяти и установить идентификаторы.

MPBLK:	.BLKW	120
MPVAR:	.WORD	MPBLK

Чтобы объявить, что идентификатор представляет число с повышенной точностью, необходимо поставить ему в соответствие блок из восьми слов в зарезервированном участке памяти с меткой **MPBLK**. Для этого можно написать макро **MP**, в результате вызова которой с переменной **VAR** будет проделана требуемая операция:

MP

VAR

Подходящим макроопределением могло бы быть

```

      .MACRO      MP      X
      ADD      #20,MPVAR
      MOV      MPVAR,X
      BR      .+2
X:      .WORD      0
      .ENDM

```

(Почему мы предпочитаем не использовать фиктивную метку в команде перехода?) Обратите внимание на то, что переменная **VAR** определяется как указатель адреса слова, расположенного сразу после блока из 8 слов. В дальнейшем это позволяет эффективно применять автодекрементную адресацию.

Лучший подход состоит в том, чтобы написать макро для всех арифметических операций, которые могут понадобиться. Так, если нужно, чтобы результатом вызова

MPCLR VAR

было обнуление переменной **VAR**, то мы должны иметь

```

      .MACRO      MPCLR      X      ?L1
      MOV      X,R1
      MOV      #10,R6
L1:      CLR      -(R1)
      SOB      R0,L1
      .ENDM

```

УПРАЖНЕНИЕ. Напишите макро, которая вызывается как **MPMOV VARA,VARB**.

Рассмотрим теперь макро **MPINC**. В ней недостаточно просто увеличить на единицу слово, содержащее младшие разряды числа (имеющие наибольший адрес), потому что во всех его битах могут быть 1, и тогда нужно перенести 1 в следующее слово. Команда **INC** для этой цели не годится, так как она устанавливает признаки, подразумевая, что операнд — целое число со знаком. В нашем же случае знаковым является 15-й бит слова, в котором хранятся старшие разряды числа (которое имеет наименьший адрес); все же остальные элементы блока просто добавляются к общему представлению числа по шестнадцать разрядов. Однако нашим требованиям прекрасно удовлетворяет команда **ADD**, поскольку она устанавливает бит **C** при переносе из старшего разряда. Поэтому макро **MPINC** начнем с прибавления единицы к младшему слову. Затем, если только бит **C** оказался равным 1, увеличим следующее слово. Для этого предназначена одноадресная команда **ADC**: она добавляет значение бита **C** по соответствующему адресу.

Может случиться, что прибавление бита **C** вновь приведет к переносу. Поскольку команда **ADC** устанавливает признаки аналогично команде **ADD**, достаточно повторить предыдущее действие по отношению к следующему слову и т. д. до старшего элемента блока. Немного усовершенствовав этот алгоритм, получим

```

        .MACRO      MPINC      X      ?L1
                MOV      X,R1
                MOV      #10,R0
                SEC
L1:      ADC      -(R1)
                SOB      R0,L1
        .ENDM

```

УПРАЖНЕНИЯ. 1. Команда **SBC** вычитает значение бита **C** из своего операнда, устанавливая при этом признаки так же, как и команда **SUB**. Напишите макро **MPDEC**.

2. Напишите макро **MPASL** и **MPASR**, правильно выполняющие умножение и деление на 2. (*Указание.* Используйте бит **C** в качестве промежуточного звена при сдвиге битов между словами.)

3. Добавьте в те макроопределения, где это необходимо, проверку результата операции на арифметическое переполнение.

4*. Составьте подпрограмму для печати целого числа повышенной точности в восьмеричном формате.

Рассмотрим, как выполняется макро **MPINC**. Каждый раз после прибавления 1 мы проверяем (используя бит **C**), помещается ли результат в текущее слово или нет; в последнем случае происходит перенос 1 в следующее слово и т. д. Это в точности совпадает с правилом прибавления 1 к целому числу в обычном позиционном представлении: если результат сложения в текущем столбце слишком велик, цифру этого столбца полагаем равной нулю и «переносим» единицу на один столбец влево. Поэтому можно рассматривать целое число повышенной точности как восьмеричное число. Значение каждой цифры есть содержимое соответствующего слова, т. е. мы имеем позиционное представление с основанием, равным $2^{16}=65\,536$!

Такой взгляд на арифметику повышенной точности делает операцию сложения очень простой. Если результат сложения для данного «столбца» превышает основание, то происходит «перенос» единицы в следующий столбец. Поэтому на каждом этапе, перед тем как применить к текущему слову команду **ADD**, сначала добавляем к нему значение бита командой **ADC**. Так же как и в команде **MPINC**, необходимо учитывать, что добавление переносимой единицы может в свою очередь вызвать перенос

и т. д. Это приводит к циклу внутри основного цикла программы. Текст всего макро приведен на рис. Б.1. Заметьте, что если в результате выполнения команды **ADC** или сложения в старших

```

      .MACRO    MPADD      X,Y      ?L1,?L2
                MOV      X,R1
                MOV      Y,R2
                MOV      #10,R0
                CLC
L1:         MOV      R2,-(SP)
                MOV      R0,-(SP)
L2:         ADC      -(R2)
                SOB      R0,L2
                BVS      ERROR
                MOV      (SP)+,R0
                MOV      (SP)+,R2
                ADD      -(R1),-(R2)
                SOB      R0,L1
                BVS      ERROR
      .ENDM

```

Рис. Б. 1. Макро для выполнения операции сложения чисел повышенной точности.

разрядах устанавливается бит V, то конечный результат будет неправильным. Быть может, тогда придется переписать программу, расширив точность представления чисел (т. е. увеличив количество ячеек, отводимых под число).

УПРАЖНЕНИЯ. 1. Напишите макро **MPSUB**.

2. Напишите макро **MPTST** и **MPCMP** так, чтобы любая следующая за ними команда перехода работала корректно. Можно ли изменить макро **MPADD** и **MPSUB**, приняв во внимание извлеченный из этого опыт?

Иногда требуется перейти от обычной формы представления числа к числу с повышенной точностью. Непосредственный перенос числа в слово, отведенное под младшие цифры, и засылка 0 в остальные ячейки блока не годится, потому что число может быть отрицательным. Команда расширения знака **SXT**¹⁾ (Sign eXTeNd) сбрасывает операнд-приемник, если бит N равен нулю, и заносит в него минус единицу, если в знаковом разряде 1. Эта команда не изменяет биты N и C. Итак, чтобы содержимое ячейки **МЕМ** перевести в формат целого числа **X** с повышенной

¹⁾ Отсутствует на некоторых малых процессорах.

точностью, нужно написать

```

                                MOV      X,R1
                                MOV      #7,R0
                                MOV      MEM,—(R1)
L1:                            SXT      —(R1)
                                SOB      R0,L1

```

Читатели, обладающие определенными познаниями в области вычислительной математики, могут пойти дальше и создать собственные макро **MRMUL** и **MRDIV**. Для вывода чисел особенно необходима операция деления на десять. Если программа выполняет только такое деление, то для экономии машинного времени целесообразно завести таблицу степеней числа десять. Предположим, что нам нужно выдать на терминал положительное число. Цифра, соответствующая каждой степени десятки, начиная с наибольшей, определяется количеством вычитаний этой степени из данного числа до получения отрицательного результата. Алгоритм очень прост и в то же время довольно эффективен.

Арифметика чисел с плавающей точкой

До сих пор мы рассматривали только арифметику чисел с *фиксированной точкой*. Точка в ней указывает на *основание* системы счисления. В системе счисления с основанием, или базой, равной десяти, точка аналогична обычной десятичной точке. Так же как столбцы, расположенные слева от десятичной точки, обозначают соответствующее количество умножений на десять, столбцы, расположенные справа, обозначают число делений на десять. Так, запись D 12.34 означает $1 \times (\text{десять}) + 2 \times (\text{один}) + 3 \times (\text{одна десятая}) + 4 \times (\text{одна десятая от одной десятой})$.

Такое же изображение чисел возможно относительно любого основания, причем в приведенном описании десятка должна быть всюду заменена на величину основания. Например, O 12.34 в десятичной записи есть $1 \times (\text{восемь}) + 2 \times (\text{один}) + 3 \times (\text{одна восьмая}) + 4 \times (\text{одна восьмая от одной восьмой}) = 8 + 2 + 3/8 + 4/64$. Здесь мы использовали *восьмеричную* систему счисления.

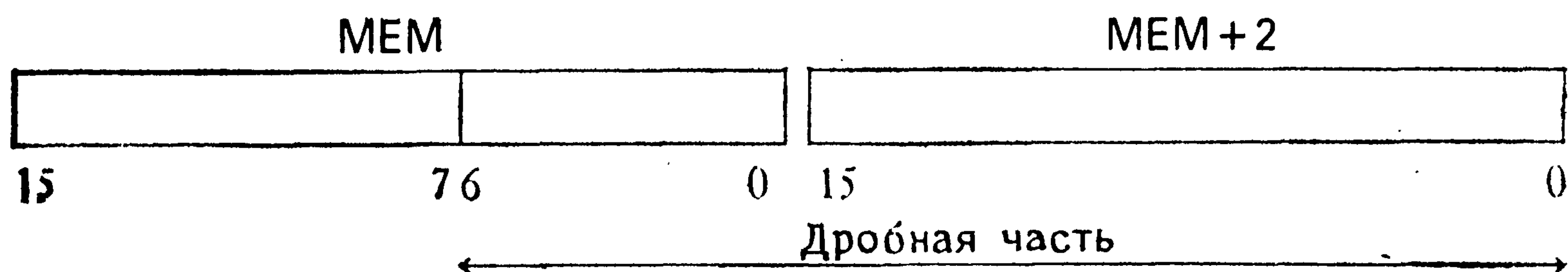
Подобным же образом можно применять *двоичную* систему. Например, B 0.10̇ = B 0.101010... = D $1/2 + 1/8 + 1/32 + \dots = D 2/3$ (вычислите сумму геометрической прогрессии)¹⁾.

УПРАЖНЕНИЕ. Найдите двоичное представление числа 1/10. Используя его, разработайте другой метод выполнения операции деления на десять.

¹⁾ Здесь используются сокращения: O от octal (восьмеричный), D от decimal (десятичный) и B от binary (двоичный). — Прим. перев.

Точка в подобной записи *фиксирована* в том смысле, что, если она присутствует, программист знает точно, в каком месте она расположена. Умножение числа D 123456 на D 234567 можно выполнить теми же машинными командами, что и умножение числа D 12345600 на D 0.0234567. В действительности же при перемножении целых чисел на ЭВМ учитываются все значащие цифры сомножителей. Иначе программа должна хранить положение точки. В предыдущем примере мы имели десятичную точку. Если же для нормализации чисел используются команды сдвига, то необходимо знать положение двоичной точки.

Макроассемблер MACRO-11 имеет обеспечение, позволяющее работать с числами в специальном *формате с плавающей точкой*, который согласован с аппаратными возможностями дополнительных плат, поставляемых к большинству процессоров. Представление чисел в плавающем формате системы PDP-11 включает три поля. Двоичная точка, отмечающая начало дробной части, располагается слева от шестого бита первого слова, где хранится число. Дробная часть располагается вправо до нулевого бита и дальше на столько слов, сколько содержит используемое представление. Последнее может быть рассчитано на одно, два или четыре слова. Так, если мы используем два слова, то имеем



Значение дробной части числа с плавающей точкой зависит от выбора порядка. Например, в десятичной записи число 123.45 можно представить как 0.12345×10^3 или как 0.012345×10^4 и т. д. до бесконечности. Первая форма записи, характеризующаяся наличием отличной от нуля цифры сразу после точки, называется *нормализованной* формой представления числа в плавающем формате. Процессоры серии PDP-11 работают исключительно с нормализованным двоичным представлением чисел. В такой записи первый бит после двоичной точки должен быть равен 1. Он *отсутствует* в машинном представлении плавающего числа (так называемый скрытый бит). Нужно подразумевать, что 6-му биту первого слова предшествует двоичное число 0.1.

Десятичные числа с плавающей точкой можно задавать ассемблеру в виде последовательности десятичных знаков, включая по необходимости десятичную точку. Порядок записывается в виде буквы E, после которой следует целое число (возможно, отрицательное или нуль). Числа в таком формате описываются директивой **.FLT2** (два слова на число) или **.FLT4** (четыре слова

на число). Например, в результате трансляции строки

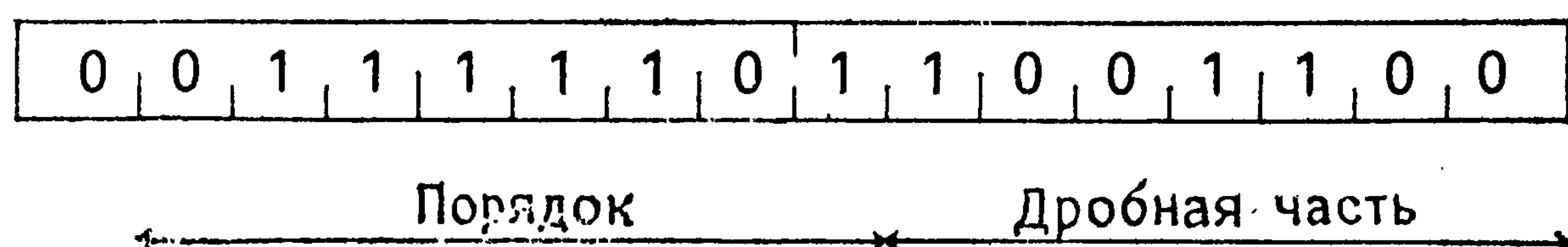
MEM: .FLT2 -2.1E-3

ассемблер заведет нормализованное двоичное представление в плавающем формате числа D —0.0021, отведя под него ячейки памяти, начиная с метки MEM.

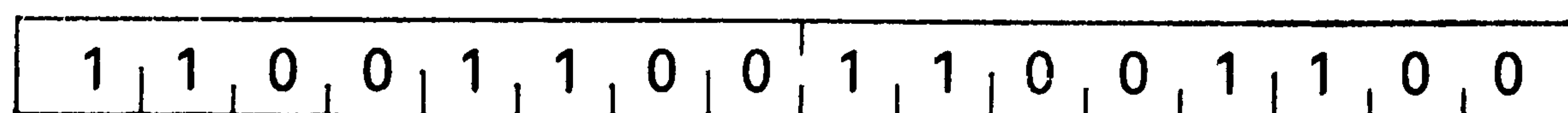
Старший бит первого слова, как и обычно, является знаковым. В битах с 7-го по 14-й кодируется двоичный порядок числа. Для положительных плавающих чисел этот код представляет собой истинный порядок числа, увеличенный на 0 200. Рассмотрим, к примеру, число D 1/10. Легко показать, что

$$\begin{aligned} D \ 1/10 &= 3/32 (1 + 1/16 + 1/64 + \dots) = \\ &= B \ 0.0001100110011\dots \\ &= 2^{-3} \times B \ 0.110011001100\dots \end{aligned}$$

Таким образом, поле порядка должно содержать 0 200—3 = 175 = B 01111101. Помня, что нужно отбросить первую единицу дробной части, получим первое слово нашего числа:



т. е. 037314. Второе слово имеет вид



или 146314. Последняя восьмеричная цифра второго слова на самом деле во время трансляции будет округлена до пяти.

Нулем считается любое число, меньшее по абсолютной величине чем D 2^{-127} , причем подразумевается, что всякое представление, содержащее нуль в поле порядка, соответствует числу с порядком —128.

Ясно, что арифметические операции с плавающей точкой должны выполняться командами, в которых учитываются особенности представления таких чисел. Возьмем, к примеру, сумму D 27+3 и посмотрим, как она вычисляется, если слагаемые записаны в плавающем формате. Используя восьмеричное представление для дробной части, имеем следующее нормализованное представление этих чисел: D 27 = O 33 = $2^5 \times O \ 0.66$; D 3 = $2^2 \times O \ 0.6$. Сначала представление числа с меньшим порядком изменяется так, чтобы порядки оказались равными. В нашем случае это приведет к записи числа 3 в виде: D 3 = $2^5 \times O \ 0.06$. (Конечно, это уже не нормализованное представление.) Теперь

выполняется сложение:

$$2^5 \times 0.66 + 2^5 \times 0.06 = 2^5 \times 0.74$$

Так же как и для их суммы, для обоих слагаемых возможен укороченный плавающий формат. Для перевода числа в этот формат в ассемблере имеется специальный оператор $\wedge F$ (стрелка вверх, затем F), за которым должно быть записано число в десятичном виде:

MOV $\wedge F27, MEM$

Программа, выполняющая рассмотренное выше сложение, могла бы начинаться со сдвига дробной части представления числа D 3 на три бита вправо для выравнивания порядков слагаемых. «Скрытый» бит при этом, конечно, должен быть представлен явно. Тогда сложение дробных частей даст корректный уже нормализованный результат.

В общем случае, однако, подобный метод не годится. Рассмотрим сложение D 27+6, причем будем считать, что оба числа представлены в укороченном плавающем формате. Поле порядка первого числа содержит 0 205. Двоичная запись дробной части, округленная до восьми разрядов, есть 0.11011000. Опуская первую единицу, получаем 7-разрядное представление дробной части: 0 130. Поле порядка числа D 6 равно 0 203, а поле дробной части в нормализованном виде — 0 100. После сдвига этого поля для выравнивания порядков на два разряда имеем в дробной части число 0 60 (не забывайте об отброшенном разряде). Сложение дробных частей приводит теперь к результату 0 210, который перекрывает поле порядка.

Подобные вычисления лучше производить с помощью *процессора плавающей арифметики*, если он входит в состав вашего ЦП. К тому же он выполняет операции существенно быстрее, чем при программной реализации. Здесь мы можем только очень коротко проиллюстрировать то, как им пользоваться.

Процессор плавающей арифметики работает только с нормализованным представлением плавающих чисел длиной в два или четыре слова и выдает результат в той же форме. Действия над этими числами выполняются особыми командами ассемблера, которые автоматически обращаются к соответствующим схемным реализациям. Так, команда **ADDF** выполняет сложение плавающих чисел длиной в два слова, а команда **ADDD** — длиной в четыре слова.

Для сложения плавающих чисел необходимо, чтобы одно из них было занесено в один из четырех *сумматоров* внутри процессора. На самом деле в него входит более четырех сумматоров, но остальные не могут использоваться для передачи дан-

ных между памятью и процессором. Обращение к ним такое же, как к первым четырем регистрам центрального процессора: аппаратура сама разбирается в том, какие регистры имеются в виду. Однако, чтобы самому не запутаться, лучше положить

F0=%0

и т. д. Также часто применяется и мнемоника **AC0** и т. п.

Проделаем вычисления, которые прежде показались бы нам трудновыполнимыми. Во-первых, устанавливаем плавающий формат слагаемых:

MEM:	.FLT2	27	
	.FLT2	6	;в MEM+4

Теперь мы должны занести одно из них на сумматор, который имеет соответствующую длину, командой загрузки плавающего числа

LDF MEM,F0

Результат сложения всегда остается на сумматоре:

ADDF MEM+4,F0

Наконец, результат можно записать в память:

STF F0,WRD

Хотя ранее расчеты на всех вычислительных машинах осуществлялись подобным образом через сумматор, сейчас уже таких машин осталось немного. Так что процессорное устройство плавающей арифметики демонстрирует всплеск ностальгии по благополучно угасающей традиции.

КОДЫ ASCII

Литера	Код
NULL	0
CONTROL-A	1
CONTROL-B	2
CONTROL-C	3
CONTROL-D	4
CONTROL-E	5
CONTROL-F	6
BELL	7
BACKSPACE	10
TAB	11
LINE FEED	12
VERT. TAB	13
FORM FEED	14
CARR. RETN	15
CONTROL-N	16
CONTROL-O	17
CONTROL-P	20
CONTROL-Q	21
CONTROL-R	22
CONTROL-S	23
CONTROL-T	24
CONTROL-U	25
CONTROL-V	26
CONTROL-W	27
CONTROL-X	30
CONTROL-Y	31
CONTROL-Z	32
ESCAPE	33
CONTROL-\	34
CONTROL-]	35
CONTROL-^	36
CONTROL-`	37
SPACE	40
!	41
"	42
#	43
\$	44
%	45
&	46
'	47
(50
)	51
*	52
+	53
,	54
-	55
.	56
/	57
0	60
1	61
2	62
3	63
4	64
5	65
6	66
7	67
8	70
9	71
:	72
;	73
<	74
=	75
>	76
?	77

Литера	Код
@	100
A	101
B	102
C	103
D	104
E	105
F	106
G	107
H	110
I	111
J	112
K	113
L	114
M	115
N	116
O	117
P	120
Q	121
R	122
S	123
T	124
U	125
V	126
W	127
X	130
Y	131
Z	132
[133
\	134
]	135
^	136
_	137
а	140
б	141
в	142
г	143
д	144
е	145
ф	146
х	147
и	150
й	151
к	152
л	153
м	154
н	155
о	156
п	157
р	160
с	161
т	162
у	163
ф	164
х	165
ц	166
ч	167
ш	170
щ	171
ъ	172
ы	173
э	174
ю	175
я	176
RUBOUT	177

СИСТЕМА КОМАНД RDP-11

DD = 6-разрядный код приемника

SS = 6-разрядный код источника

R = 3-разрядный код регистра

X = 0 для команд, занимающих слово, X = 1 для байтовых команд

Мнемоника	Код	Описание	Изменение разрядов условий
CLR (B)	X050DD	Сбрасывает признак	N, V, C = 0; Z = 1
DEC (B)	X053DD	Вычитает 1 из признака	N, Z, V = 0/1 по результату
INC (B)	X052DD	Прибавляет 1 к признаку	N, Z, V = 0/1 по результату
NEG (B)	X054DD	Меняет знак признака	N, Z, V = 0/1 по результату
TST (B)	X057DD	Устанавливает разряды условий	C = 0, если результат 0, иначе C = 1 N, Z = 1/0 по содержимому признака V, C = 0
COM (B)	X051DD	Образует дополнение к признаку	N, Z = 1/0 по результату, V = 0, C = 1
ASR (B)	X062DD	Сдвиг признака на одну позицию вправо; старший бит дублируется	N, Z = 1/0 по результату C ← старший младший бит признака V ← <i>исключающее или</i> битов N и C
ASL (B)	X063DD	Сдвиг признака на одну позицию влево; в младший бит помещается 0	N, Z = 1/0 по результату C ← старший старший бит признака V ← <i>исключающее или</i> битов N и C
ADC (B)	X055DD	Прибавляет бит C к признаку	1/0 по результату (для всех)
SBC (B)	X056DD	Вычитает бит C из признака	1/0 по результату (для всех)
SXT	0067DD	Все разряды признака по значению бита N	Z = 1, если бит N = 0, V = 0
ROR (B)	X060DD	Циклический сдвиг признака вправо на 1 бит через бит C	N, Z = 1/0 по результату C ← старший младший бит признака V ← <i>исключающее или</i> битов N и C
ROL (B)	X061DD	Циклический сдвиг признака влево на 1 бит через бит C	N, Z = 1/0 по результату C ← старший старший бит признака V ← <i>исключающее или</i> битов N и C
SWAB	0003DD	Перестановка байтов признака	N, Z = 1/0 по старому значению признака, V, C = 0

ОДНОАДРЕСНЫЕ

Ветвление: $PC \leftarrow PC + (2 \times \text{смещение})$. Кодировается: код + смещение. Разряды условий не меняются.

Мнемоника	Код	Условие	Мнемоника	Код	Условие
BR	000400	Всегда			
BEQ	001400	$Z=1$	BNE	001000	$Z=0$
BMI	100400	$N=1$	BPL	100000	$N=0$
BCS	103400	$C=1$	BCC	103000	$C=0$
BVS	102400	$V=1$	BVC	102000	$V=0$
BLT	002400	$N \vee V=1$	BGE	002000	$N \vee V=0$
BLE	003400	$Z \vee (N \vee V)=1$	BGT	003000	$Z \vee (N \vee V)=0$
BLOS	101400	$C \vee Z=1$	BHI	101000	$C=0$ и $Z=0$

ВЕТВЛЕНИЯ

Мнемоника	Код	Описание	Изменение разрядов условий
MOV (B)	X1SSDD	Пересылает источник в признак	$N, Z=1/0$ по результату, $V=0$
ADD	06SSDD	Складывает источник и признак	$1/0$ по результату (для всех)
SUB	16SSDD	Вычитает источник из признака	$1/0$ по результату (для всех)
CMP (B)	X2SSDD	Образует (источник — признак)	$1/0$ по результату (для всех)
BIS (B)	X5SSDD	Помещает (источник или признак) в признак	$N, Z=1/0$ по результату, $V=0$
BIT (B)	X3SSDD	Образует (источник и признак)	$N, Z=1/0$ по результату, $V=0$
BIC (B)	X4SSDD	Помещает (\sim источник и признак) в признак	$N, Z=1/0$ по результату, $V=0$
MUL	070RSS	Умножение } результат в R и в	$1/0$ по результату (для всех)
DIV	071RSS	Деление } след. рег., если R нечетный	$1/0$ по результату (для всех)
XOR	074RDD	Исключающее или, результат в признак	$N, Z=1/0$ по результату, $V=0$

ДВУХАДРЕСНЫЕ

Мнемоника	Код	Описание	Мнемоника	Код	Описание
JSR	004RDD	Регистр \rightarrow стек, PC \rightarrow регистр, признак \rightarrow PC			
RTS	00020R	Регистр \rightarrow PC, стек \rightarrow регистр			
SPL	00023L	Устанавливает приоритет ЦП в L			
JMP	0001DD	Признак \rightarrow PC			
SOB	077RXX	XX=смещение; вычитает 1 из содержимого регистра; если $\neq 0$, то переход назад			
EMT	104000—104377	PS, PC \rightarrow стек, новое PC, PS из 30, 32			Все загружаются из 32
TRAP	104400—104777	PS, PC \rightarrow стек, новое PC, PS из 34, 36			Все загружаются из 36
BPT	000003	PS, PC \rightarrow стек, новое PC, PS из 14, 16			Все загружаются из 16
RTI	000002	Загружает PC, PS из стека			Все загружаются из стека
RTT	000006	PC, PS из стека, откладывает трассир. прер.			Все загружаются из стека
MFPI	0065SS	Пересылает слово из предшествующего пространства в текущий стек			N, Z=1/0 по значению источника, V=0
MTPI	0066DD	Пересылает слово из текущего стека в предшествующее пространство			N, Z=1/0 по результату, V=0
HALT	000000	Останавливает ЦП			
WAIT	000001	Ожидает прерывания			
RESET	000005	Инициализирует общую шину			
РАЗРЯДЫ УСЛОВИЙ					
CLC	000241	Сбрасывает бит C	SEC	000261	Устанавливает бит C
CLV	000242	Сбрасывает бит V	SEV	000262	Устанавливает бит V
CLZ	000244	Сбрасывает бит Z	SEZ	000264	Устанавливает бит Z
CLN	000250	Сбрасывает бит N	SEN	000270	Устанавливает бит N
CCC	000257	Сбрасывает все биты	SCC	000277	Устанавливает все биты

УКАЗАТЕЛЬ МАКРОКОМАНД В АССЕМБЛЕРЕ MACRO-11

Стоящая вначале точка означает, что это псевдооператор

ADC (B) 252

ADD 37

.ASCII, .ASCIZ 107, 109

.ASECT, .CSECT 125, 126

ASL (B), ASB (B) 139

BCC, BCS 135

BGE 133

BHIS 135

BIC (B), BIS (B), BIT (B) 127—129

BLE 134

.BLKB, .BLKW 73, 125

BLO 135

BLT 133

BPL 59, 132

BPT 157

BR 61

BVC, BVS 132

CCC, CLC, CLN, 136

CLR (B) 54, 136

CLV, CLZ 136

CMP (B) 63, 120, 137

COM (B) 129

DEC (B) 60

DZV 53, 249

EMT 154

.ENABL, .DSABL 153, 154

.END 41

.ENDC 166

.ENDM 160

.EVEN 110

.GLOBL 239

HALT 70

.IF, .IFF, .IIF 166, 168, 170

INC 36

.IRP, .IRPC 207, 208

JMP 155

JSR 98

.LIST, .NLIST 45, 207

.MACRO 160

.MCALL 42

MFPI, MTPI 235

MOV (B) 33, 108, 137

MUL 54, 248

NEG (B) 62

.NTYPE 170

.RADIX 165

.RAD50 200

RESET 222

ROL (B), ROR (B) 140

RTI 152

RTS 103

RTT 197

SBC (B) 252

SCC, SEC, SEN, SEV, SEZ 136

SOB 142

SPL, MTPS 193, 194

SUB 37

SWAB 138

SXT 253

.TITLE 45

TRAP 156

TST (B) 64

WAIT 196

.WORD, .BYTE 70, 109

XOR 129

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- Автоматическое продвижение указателя 76
- Адрес абсолютный (absolute address) 46
 - базовый (base) 227
 - виртуальный (virtual) 224
 - исполнительный (effective) 81
 - — вычисление 81
 - начальный (start) 40
 - относительный (relative) 45, 92
 - передачи управления 46
 - перемещаемый (relocatable) 76, 83, 219
 - слова (word) 35
 - физический (physical) 224
 - — формирование 227
- Адресации режим (addressing mode) 81
 - — относительный 91
- Адресация (addressing) 33
 - абсолютная 153
 - автодекрементная (autodecrement) 77
 - автоинкрементная (autoincrement) 77
 - косвенная (indirect) 84
 - — разряд (bit) 84
 - непосредственная (immediate mode) 89
 - с использованием счетчика команд 89
- Адресов счетчик (location counter) 124
- Аппаратные средства (hardware) 11, 187
- Аппаратный предзагрузчик (bootstrap) 181
 - указатель стека (stack pointer) 121
- Арифметика (arithmetic) 248
 - целая повышенной точности 249
 - чисел с плавающей точкой (floating point) 254
 - — — фиксированной точкой (fixed point) 254
- Арифметический подход 127
- Арифметических выражений вычисление 114
- Ассемблер (assembler) 37, 39, 98
- Ассемблера макродиректива (macro directive) 207
 - язык (language) 40
- Базовый адрес 227
- Байт (byte) 34
 - младший (low) 34
 - старший (high) 34
- Байтовые команды (byte instructions) 136
- Бит включения управления памятью 225
 - ошибки объединенный 222
 - пусковой (go bit) 216
 - сброса контроллера (controller clear) 221
 - сброшенный (clear) 28
 - скрытый (hidden) 255
 - установленный (set) 28
 - C 134
 - N 64
 - T 196
 - V 131
 - Z 65
- Бита состояние (state) 27
- Биты (bits) 27
- Блок (block) 206
- Буфер (buffer) 18, 193, 208
 - ввода (input) 50
 - печатающего устройства терминала 35
 - сохранения (save) 26
- Буфера заполнение 209
- Буферизация ввода-вывода 208
 - многократная 210
- Буферов кольцо 211
- Буферы вспомогательные 26
- Ввод команды 19
 - литер (character input) 178
 - под управлением монитора 50
 - чисел 56, 61

- Ввода буфер 50
Ввода-вывода буферизация 208
Ввод-вывод с двойной буферизацией 193
— с терминала и пульта управления 174
— управляемый по прерываниям (interrupt-driven I/O) 190
Вектор прерывания 149, 186
Ветвление (условный переход) (branch) 59
Виртуальный адрес 224
Включение управления памятью 225
Вложение подпрограмм 102
— прерываний 195
Вложенные подпрограммы (nesting subroutines) 99
Внешние запоминающие устройства под управлением монитора 198
— — — — — пользователя 212
Внутренний регистр (слово состояния процессора) 64
Восьмеричная система счисления 254
Вспомогательные буферы 26
Вставка (insert) 19
— заплат (patching) 156
Выбор основания системы счисления 165
Вывод, управляемый монитором 38
— чисел 53
Вызов параметра по значению 147
— — — результату 147
— — — ссылке 148
Вызова последовательность 96
Вызовы мониторные 198
Выполнение программы (program execution) 38, 244
Выходные данные (output) 38
Вычисление арифметических выражений 114
Вычислительная система (computer system) 11
- Глобальные имена 239
Глубина (уровень) вложенности (depth (level) of nesting) 102
Головка (head) 213
— только для чтения 213
— чтения-записи 213
- Данные выходные 38
Данных источник 49
— приемник 49
Двоичная система счисления 28, 254
Двоичное дополнение (twos complement) 93, 218
Двухадресные команды 85
- Деление (division) 53, 248
— на 10 59
Десятичная система счисления 165, 254
Директивы повторения 207
Диски (disks) 11
Дисков пакет (disk cartridge) 212
Дисковая система RK06 212
Дисковод (drive) 215
Дисковод контроллер 212
Дополнение логическое 129
Дорожки (tracks) 213
Доступа права 228
Драйверы устройств (device handlers) 201
- Завершающий фрагмент (completion routine) 212
Заголовок (header) 216
Загрузки карта 47
Загрузчик (loader) 40
Загрузчики начальные (bootstraps) 180
Задание (job) 16
Закрытие выходного файла (close a file) 20
Запись в память (write into memory) 27
— в файл 18, 205
— на диск и чтение с диска 217
Заплаты (patches) 157
— вставка 156
Заполнение буфера с диска 209
Запоминание и выборка информации 27
Запрос на прерывание (interrupt request) 186
Запроса канал 186
— обслуживание (service) 186
Защита памяти 230
Знаковый разряд (sign bit) 93
- Изолированная система (standalone system) 11
Имена глобальные (global symbols) 239
— локальные (local) 163
Индексация (indexing) 71
Индексный регистр (index register) 71
— режим (mode) 81
Интерфейс (interface) 187
Информации выборка (retrieving information) 27
— запоминание (storing) 27
Исполнительный адрес, вычисление 81

- Источник (source) 49, 85
- Канал запроса (bus request) 186
- Карта загрузки (load map) 47
- Квант (block) 228
- Код позиционно-независимый (position independent code) 92
 - 50-ричный (radix-50) 199
- Коды команд перехода (encoding branch instructions) 141
 - условий (condition codes) 64
- Кольцо буферов 211
- Команд регистр (instruction register) 35
 - счетчик (counter) (PC) 36
 - типы (types) 42
- Команды арифметического и циклического сдвигов 138
 - байтовые (byte instructions) 136
 - ввод (input) 19, 20
 - двухадресные (double-operand) 85
 - логические (logical) 126
 - контекстного поиска (character-oriented commands) 23
 - одноадресные (single-operand) 81
- Команды перехода (branch instructions) 58
 - — безусловного 61
 - — условного 59
 - позиционно-независимые (position independent) 153
 - синтаксис (syntax) 19
 - MUL и DIV 248
- Комментарии (comments) 57
- Компиляция (compile) 44
- Компоновка 238
- Компоновщик (linker) 41, 46
- Константа перемещения (relocation constant) 76
- Контроллер (controller) 215
 - дисковод (drive) 212
- Косвенная адресация 84
- Ленты (tapes) 11
- Листинг (listing) 75
 - программы 240
 - файла 44
- Литер ввод (character input) 179
 - вывод последовательный (output, sequential) 177
- Литера ограничительная (separator character) 61
- Литерал (literal) 146
- Литеры эхо 51
- Логические команды 126
- Логические ошибки 89
- Логическое дополнение (logical complement) 129
- Локальные имена 163
 - метки 163
- Макро (macros) 160
 - вложенные (nesting) 164
 - параметры (arguments) 161
 - тело (body) 160
- Макробιβлиотека системная (system macros library) 44
- Макровывоз (macro call) 160
- Макродиректива ассемблера 207
- Макроимена (macro labels) 162
- Макрокоманда (macro) 40, 158
 - расширение (expand) 160
 - системная (system) 42
- Макроопределение (definition of macro) 160
- Метки (labels) 41
 - локальные (local) 110
- Метод сортировки (sorting technique) 190
- Минус бинарный (minus binary) 117
 - унарный (unary) 117
- Многokратная буферизация (multiple buffering) 210
- Модули (modules) 239
- Модульная система 14
- Модульное программирование 143
- Монитор (monitor) 13
- Мониторная программа 192
- Мониторные вызовы (monitor calls) 198
- Начальные загрузки 180
- Начальный адрес 40
- Непосредственная адресация 89
- Нормализованная форма представления числа 255
- Обслуживание запроса 186
- Общая шина (UNIBUS) 185
- Ограничители (delimiters) 107
- Операнд (operand) 40
- Оперативный стек 236
- Оператор присваивания (assignment statement) 48
- Оператора терминал 2, 234
- Операционная система (operating system) 13
- Операция и (ana) 127

- Операция или включающее (*inclusive or*) 128
— — исключающее (*exclusive or*) 128
— не (*not*) 129
Организация памяти страничная 224
Основание системы счисления 28
Открытие файла 17, 202
Отладка и исправление программы 240
Отладки средства (*debugging aids*) 157
Относительный адрес 45, 92
— режим адресации 91
Отрицательные числа 93
Ошибка синхронизации (*timing error*) 222
— фазы трансляции (*phase error*) 173
Ошибки (*errors*) 221
— логические (*logical*) 89
— объединенный бит (*combined*) 222
— синтаксические (*syntax*) 88
- Пакет дисков 212
Памяти защита 230
Память (запоминающее устройство) (*memory, storage area*) 19, 27, 69
— на ферритовых сердечниках (*magnetic core*) 27
Параметров передача (*passing parameters*) 143
Параметры в языках высокого уровня 145
— макро 161
— формальные 161
Перемещаемый адрес 76, 83, 219
Перемещения константа 76
Перенос (*carry*) 134
Переполнение (*overflow*) 130
Переходы при условной трансляции 169
Периферийные устройства (*peripheral devices*) 11
Пластины (*platters*) 212
Подпрограмм вложение 102
— связь 102
Подпрограмма обработки прерываний от клавиатуры 187
Подпрограммы (*subroutines*) 95
— вложенные 99
— вызов (*calling*) 96
— рекурсивные (*recursive*) 117
Подсчет элементов данных 66
Позиционно-независимые команды 153
Позиционно-независимый код 92
- Поле смещения (*displacement field*) 227
— текущей страницы (*active page*) 227
Поразрядная четность (*parity*) 138
Последовательность вызова 96
Последовательный вывод литер 177
Права доступа (*access rights*) 228
Предзагрузчик аппаратный (*hardware bootstrap*) 181
— программный (*software*) 181
Прерывание висящее (*pending interrupt*) 196
— (внешнее) (*interrupt*) 196
— трассировочное (*trace*) 158, 196
Прерываний вложение (*nesting*) 195
— разрешение (*enabling*) 187
Прерывания (*interrupts*) 184, 219
— вектор 149, 186
— для печатающего устройства (*terminal printer*) 190
— от дисководов и контроллера 220
— программно-генерируемые (*program-generated traps*) 154
— программные (*traps*) 148
— — программа обслуживания 149, 150
Приемник (*destination*) 49, 85
Приоритет (*priority*) 186, 193
— ЦП (*CPU*) 193
Приоритета уровень 193
Присваивания оператор 48
Программа (*program*) 16
— мониторная 192
— обслуживания программного прерывания 149, 150
— отладка и исправление 240
— редактирования текста (*text-editing*) 96
Программирование модульное 143
Программное обеспечение (*software*) 13
Программы ветвящиеся (*branch programs*) 95
— выполнение 38, 244
— листинг 240
— системные 13
— структурная разработка 158
Процесс трансляции (*assembly process*) 43
Процессор плавающей арифметики (*floating point processor*) 257
— центральный 35
Псевдооператоры (*pseudo-ops*) 42
Пультовые переключатели (*console switches*) 175
Пусковой бит 216

- Разделения времени режим (time-sharing) 11
 — — системы (systems) 16
 Разделитель (separator character) 160
 Размер страницы (page length) 228
 Разметка (formatting) 216
 Разрешение прерываний 187
 Разряд знаковый 93
 — косвенной адресации 84
 — разрешения прерываний (interrupt enable bit) 219
 Разряды (биты) условий 64
 Ракорд (leader) 183
 Распечатка файла 44
 Расширение имени файла (extension of file name) 17
 — — — по умолчанию (default) 18
 — макрокоманды 160
 Регистр (register) 48, 98
 — адреса диска 218
 — — страницы 225
 — — шины 218
 — индексный (index) 71
 — команд 35
 — описания страницы 225
 — ошибок дисководов 222
 — связи (linkage) 102
 — состояния дисководов 215
 — — печатающего устройства 178
 — — управления памятью 225
 — счетчика слов 218
 — требуемого цилиндра 216
 — управления и состояния 214
 Регистры смещения (relocation registers) 240
 — устройства (device) 214
 Редактирование файла (editing a file) 20
 — текста 96
 Редактор (editor) 16
 Режим адресации (addressing mode) 81
 — индексный (index) 81
 — команд монитора 17, 21
 — обычный (пользовательский) (user) 231, 234
 — оперативный (kernel) 231, 235
 — пошаговый (single-instruction) 246
 — — отмена 247
 — работы процессора (processor) 231
 — разделения времени (timesharing) 11
 — регистровой адресации (addressing mode 0) 236
 — off-line (автономный) 12
 — on-line (оперативный) 12
 Рекурсивные подпрограммы 117
 Связанный список (linked list) 86
 Связь между адресными пространствами 234
 — подпрограмм (subroutine linkage) 102
 Сдвиг арифметический (shift) 138
 — циклический (rotate) 138
 Сектор (sector) 216
 Синтаксис команды 19
 Синтаксические ошибки 88
 Система вычислительная (computer system) 17
 — дисковая (disk) 212
 — модульная (modular) 14
 — операционная 13
 — счисления восьмеричная (octal notation) 29, 254
 — — двоичная (binary) 28, 254
 — — десятичная (decimal) 28, 165, 254
 Системная макробиблиотека 44
 — макрокоманда 42
 Системные программы (systems programs) 13
 Системы разделения времени 16
 — счисления основание (base (radix)) 28, 165
 Слова содержащее (contents of word) 28
 — формат (format) 81
 Слово состояния процессора (внутренний регистр) (processor status word) 64
 Слово-заголовок (header-word) 210
 Сопрограммы (coroutines) 122
 Сортировка (sorting) 78
 Сортировки метод 190
 Состояния бита 27
 Справочник файлов (directory of files) 20
 Средства отладки 157
 Стек (stack) 111
 — оперативный (kernel) 236
 — системный (system) 121, 235
 Стека указатель (stack pointer) 112
 Страница (page) 25, 224
 — нерезидентная (nonresident) 229
 — резидентная, открытая на чтение-запись 229
 — — — только на чтение 229
 Страницы размер и доступ к ней 228
 — текущей поле 227
 Страничная организация (page structure) 25

- Страничная организация памяти (memory pages) 224
Строка (line) 21
Структурная разработка программ 158
Ссылки вперед (forward references) 171
Сумматоры (accumulators) 257
Счетчик адресов 124
— команд 36
— проходов (proceed) 247
Считывание из памяти (read from memory) 27
- Таблица распределения памяти (symbol table) 70
Тело макро 160
Текст ASCII 106
Терминал (terminal) 11
— оператора (console) 234
Терминальный буфер вывода (terminal printer buffer) 174
Типы команд 42
Точность двойная (double precision) 249
Точки останова (break points) 245
Трансляции процесс 43
Трансляция условная (conditional assembly) 166
— — переходы 169
Трассировка (tracing) 75
Трассировочное прерывание 158
Триада (triad) 30
Триггеры (flip-flops) 27
- Указателя продвижение автоматическое 76
Указатель (pointer) 22, 72, 84
— стека 112
— (регистр) связи 102
Умножение (multiplication) 54, 248
— на 10 60
Унарный минус 117
Управление выполнением команд 126
— памятью (memory management) 222
— — включение (enabling) 225
Уровень (глубина) вложенности 102
— приоритета 193
Условий коды 64
— разряды (биты) 64
Условия, включающие выражения 168
Условная трансляция 166
— — переходы 169
- Условный переход (ветвление) 59
Установка условных признаков 136
- Файл (file) 15
— запись (writing) 19, 205
— исходный (source) 43
— образа памяти (memory image) 46
— объектный (object) 43
— открытый для ввода (opening for input) 20
— — — вывода (opening for output) 18
Файла выходного закрытие 20
— листинг 44
— открытие 17, 202
— распечатка 44
— редактирование 20
— чтение (reading) 209
Файлов справочник 20
Физический адрес 224
— — формирование 227
Формальные параметры 161
Формат с плавающей точкой 255
— слова 81
Фрагмент завершающий 212
- Центральный процессор (ЦП) (central processing unit (CPU)) 35
Цифры (numerals) 28
- Четность поразрядная 138
Чтение файла 209
Чисел ввод (numerical input) 56, 61
— вывод (output) 53
Числа отрицательные (negative numbers) 93
- Шина общая 185
- Эмулятор пульта управления (console emulator) 176
Эхо литеры (echo) 51
- Язык ассемблера 40

ALTMODE 18

LINK 46
LSI-11 16

BACKSPACE 16

MACRO-11 40

CARRIAGE RETURN 16, 17, 25
CONTROL 11
CONTROL-литеры 12

n-переключатель 155

DELETE 16
DDT 75ODT-11 75
ODT работа 238

PAL-11 40

EDIT 17
ESCAPE 18RSTS 46
RT-11 38, 46
RSX-11 46
RUBOUT 16

FORM FEED 26

SHIFT 11

IAS 46

TECO 18

LIFO 111
LINE FEED 19

UNIX 15

ОГЛАВЛЕНИЕ

Предисловие редактора перевода	5
Предисловие	7
1. Введение	11
1.1. Вычислительная система	11
1.2. Редактор	16
1.3. Запоминание и выборка информации	27
1.4. Выполнение программы	38
2. Основы	48
2.1. Регистры	48
2.2. Команды перехода	58
2.3. Память	68
2.4. Формат слова	81
3. Структура программы	95
3.1. Подпрограммы	95
3.2. Стеки	111
3.3. Управление выполнением программы	126
3.4. Модульное программирование	143
3.5. Структурная разработка программы	158
4. Периферийное оборудование	174
4.1. Ввод-вывод с терминала и пульта управления	174
4.2. Прерывания	184
4.3. Внешние запоминающие устройства под управлением монитора	198
4.4. Внешние запоминающие устройства под управлением пользователя	212
4.5. Управление памятью	222
Приложение А. ODT	238
Работа ODT	238
Глобальные имена	239

Отладка и исправление программы	240
Исполнение программы	244
Приложение Б. Арифметика	248
Команды MUL и DIV	248
Целая арифметика повышенной точности	248
Арифметика чисел с плавающей точкой	254
Коды ASCII	259
Система команд PDP-11	260
Указатель макрокоманд в ассемблере MACRO-11	263
Предметный указатель	264

Майкл Сингер

МИНИ-ЭВМ PDP-11: ПРОГРАММИРОВАНИЕ
НА ЯЗЫКЕ АССЕМБЛЕРА И ОРГАНИЗАЦИЯ
МАШИНЫ

Старший научный редактор И. А. Маховая
Младший научный редактор Н. С. Полякова
Художник Н. Я. Бовк
Художественный редактор В. И. Шаповалов
Технический редактор Т. А. Максимова
Корректор М. А. Смирнов

ИБ № 3909

Сдано в набор 24.02.84. Подписано к печати 20.07.84. Формат 60×90^{1/16}. Бумага
типографская № 2. Гарнитура литературная. Печать высокая. Объем 8,50 бум. л.
Усл. печ. л. 17. Усл. кр.-отт. 17,52. Уч.-изд. л. 15,28. Изд. № 1/3226. Тираж 50 000 экз.
Заказ № 328. Цена 1 р. 10 к.

ИЗДАТЕЛЬСТВО «МИР» 129820, Москва, И-110, ГСП, 1-й Рижский пер., 2

Отпечатано в Ленинградской типографии № 2 головном предприятии ордена Трудового
Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении
Соколовой Союзполиграфпрома при Государственном комитете СССР по делам
издательств, полиграфии и книжной торговли. 198052, г. Ленинград, Л-52, Измайлов-
ский проспект, 29 с матриц ордена Октябрьской Революции и ордена Трудового
Красного Знамени Первой Образцовой типографий имени А. А. Жданова Союзполи-
графпрома при Государственном комитете СССР по делам издательств, полиграфии
и книжной торговли. 113054, Москва, Валовая, 28

**КНИГИ,
ВЫПУЩЕННЫЕ ИЗДАТЕЛЬСТВОМ «МИР»
В СЕРИИ**

«МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ»

- Баррон Д. Ассемблеры и загрузчики.— 1974.— 5 л.
Баррон Д. Рекурсивные методы в программировании.— 1974.— 5 л.
Коддингтон Л. Ускоренный курс Кобола.— 1974.— 17 л.
Фостер Дж. Обработка списков.— 1974.— 4,5 л.
Хенли Дж. Автоматизированная библиотека и информационные системы.— 1974.— 7,5 л.
Хигман Б. Сравнительное изучение языков программирования.— 1974.— 13 л.
Дал У., Дейкстра Э., Хоор К. Структурное программирование.— 1975.— 15 л.
Джадд Д. Работа с файлами.— 1975.— 9 л.
Колин А. Введение в операционные системы.— 1975.— 7,5 л.
Фостер Дж. Автоматический синтаксический язык.— 1975.— 3,5 л.
Хамби Э. Программирование таблиц решений.— 1975.— 5 л.
Грунд Ф. Программирование на языке Фортран-IV.— 1976.— 10 л.
Маурер У. Введение в программирование на языке Лисп.— 1976.— 6 л.
Браун П. Макропроцессоры и мобильность программного обеспечения.— 1977.— 18 л.
Вайнгартен Ф. Трансляция языков программирования.— 1977.— 14 л.
Вирт Н. Систематическое программирование. Введение.— 1977.— 10 л.
Оллонгрэн А. Определение языков программирования.— 1977.— 17 л.
Дейкстра Э. Дисциплина программирования.— 1978.— 13 л.
Хигман Б. Введение в программирование для вычислительных машин.— 1978.— 5 л.
Холл П. Вычислительные структуры. Введение в численное программирование.— 1978.— 9 л.
Пейган Ф. Практическое руководство по Алголу-68.— 1979.— 10 л.
Пересмотренное сообщение об Алголе-68.— 1979.— 34 л.
Баррон Д. Введение в языки программирования.— 1980.— 9 л.
Грисуолд Р., Поудж Дж., Полонский И. Язык программирования Снобол-4.— 1980.
Турский В. Методология программирования на ЭВМ.— 1981.— 16 л.

1 р. 10 к.

МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ

- Мак-Кранен Д. Руководство по программированию на языке PL/M для микро-ЭВМ.— 1981.— 10 л.
- Хьюз Ч., Пфлигер Ч., Роуз Л. Методы программирования. Курс на основе Фортрана.— 1981.— 20 л.
- Эшли Р., Фернандес Д. Н. Язык управления заданиями.— 1981.— 10 л.
- Алгоритмический язык Алгол-60. Модифицированное сообщение.— 1982.— 5 л.
- Грегора П. Программирование на языке Паскаль.— 1982.— 18 л.
- Данные в языках программирования. Абстракция и типология. Сб. статей.— 1982.— 22 л.
- Кацан Г. Язык Фортран-77.— 1982.— 12 л.
- Вегнер П. Программирование на языке АДА. Введение.— 1983.— 12 л.
- Хендерсон П. Функциональное программирование. Применение и реализация.— 1983.— 21 л.
- Деннинг В., Эсик Р., Маас С. Диалоговые системы «Человек — ЭВМ». Адаптация к требованиям пользователя.— 1984.— 8 л.
- Сингер М. Мини-ЭВМ PDP-11. Программирование на языке ассемблера и организация машины.— 1984.— 16 л.
- Требования и спецификации в разработке программ. Сб. статей.— 1984.— 20 л.

В 1985 г.
В ИЗДАТЕЛЬСТВЕ «МИР»
В СЕРИИ
«МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ»
ГОТОВЯТСЯ К ВЫПУСКУ

- Варт Н. Алгоритмы + структуры данных — программы.— 1985.— 19 л.— 2 р.
- Койллингерт П. Элементы операционных систем.— 1985.— 20 л.— 1 р. 70 к.
- Меткалф М. Оптимизация в Фортране.— 1985.— 15 л.— 1 р. 20 к.
- Фокс Дж. Программное обеспечение и его разработка.— 1985.— 20 л.— 2 р. 30 к.

